Trabajo Fin de Grado

Ingeniería Electrónica, Robótica y Mecatrónica

Ensayo de técnicas de localización y seguimiento visual para vehículos aéreos no tripulados

Autor: Iván de la Fuente Trinidad

Tutor: Carlos Vivas Venegas

Dpto. Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado

Ingeniería Electrónica, Robótica y Mecatrónica

Ensayo de técnicas de localización y seguimiento visual para vehículos aéreos no tripulados

Autor:

Iván de la Fuente Trinidad

Tutor:

Carlos Vivas Venegas

Profesor titular

Dpto. de Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado: Ensayo de técnicas de localización y seguimiento visual para vehículos aéreos no tripulados

Autor: Iván de la Fuente Trinidad

Tutor: Carlos Vivas Venegas

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A mi familia

A mis abuelos

A mis amigos

Agradecimientos

La universidad ha sido una etapa que me ha marcado en todos los sentidos.

Quería agradecerle en primer lugar a mi familia por dármelo todo y más para conseguir mis objetivos. Por apoyarme con mis decisiones, aconsejarme y estar en los momentos más difíciles. Sin ellos no hubiera sido posible. Son lo más importante.

A mis compañeros, en especial a Paula, Juan y Ana, por todos esos momentos dentro y fuera de la universidad. Gracias por ayudarme siempre y hacer esta etapa inolvidable.

A todos mis amigos, por estar ahí siempre.

A Carlos Vivas por ofrecerme trabajar en este proyecto.

A todos mis profesores por acompañarme en este largo trayecto de aprendizaje.

Iván de la Fuente Trinidad

Sevilla, 2021

Resumen

Los vehículos aéreos no tripulados (UAV) han demostrado recientemente un gran rendimiento en la recopilación de datos visuales a través de la exploración y el mapeo autónomos, que se utilizan ampliamente en aplicaciones de reconocimiento, vigilancia y adquisición de objetivos (RSTA).

El objetivo principal de este proyecto es el ensayo mediante simulación de un sistema basado en visión a bordo para UAV de bajo costo para rastrear de forma autónoma un objetivo en movimiento. Se planteará el seguimiento visual en tiempo real mediante el uso de un algoritmo de detección de objetos llamado SIFT. Para apuntar al objetivo seleccionado durante los vuelos se utiliza una cámara con cardán de 3 ejes con unidad de medida inercial (IMU).

Para la simulación se hará uso de Gazebo, un simulador de entornos 3D, de código abierto, que permite evaluar el comportamiento de robots en un mundo virtual.

En este proyecto se parte del trabajo realizado por Isidro Marcelo Jáñez Vaz en su TFG ([1]). Creó un entorno de simulación, donde incluyó un modelo 3D del sistema diseñado, formado por un quadrotor, un gimbal de tres ejes y una cámara. También resolvió el control de vuelo de dicho sistema y el control en posición del gimbal.

ROS, un entorno de trabajo gratuito para el desarrollo de software para robots, será la herramienta utilizada para la implementación del algoritmo de detección y del control del gimbal para que este realice el seguimiento del objetivo deseado.

Abstract

Unmanned aerial vehicles (UAVs) have recently demonstrated high performance in visual data collection through autonomous scanning and mapping, which are widely used in reconnaissance, surveillance and target acquisition (RSTA) applications.

The main objective of this project is to perform a simulation test of an on-board vision-based system for low-cost UAVs to autonomously track a moving target. Real-time visual tracking will be achieved by using an object detection algorithm called SIFT. A 3-axis gimbal camera with an inertial measurement unit (IMU) is used to point the selected target during flights.

For the simulation we will make use of Gazebo, an open-source 3D environment simulator, which allows to evaluate the behavior of robots in a virtual world.

In this project we start from the work done by Isidro Marcelo Jáñez Vaz in his TFG. He created a simulation environment, where he included a 3D model of the designed system, consisting of a quadrotor, a three-axis gimbal and a camera. The flight control of the system and the position control of the gimbal are also solved.

ROS, a free framework for the development of software for robots, will be the tool used to implement the detection algorithm and the control of the gimbal to track the desired target.

Índice

ÍNDICE DE CÓDIGOS

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

$\times$	Operador de producto vectorial
$\cdot$	Operador de producto escalar
S	Función que transforma vector en matriz antisimétrica
S^{-1}	Función que transforma matriz antisimétrica en vector
$\dot{A}$	Derivada con respecto al tiempo de la matriz A
A'	Traspuesta de la matriz/vector A
$\|\| a \|\|$	Norma del vector a
A^{-1}	Matriz inversa de A

1 INTRODUCCIÓN

1.1. Motivación

Un vehículo aéreo no tripulado (VANT), UAV (del inglés unmanned aerial vehicle), más apropiadamente RPAS (del inglés Remotely Piloted Aircraft System), comúnmente conocido como dron, hace referencia a una aeronave que vuela sin tripulación, la cual ejerce su función remotamente. Un VANT es un vehículo sin tripulación, reutilizable, capaz de mantener de manera autónoma un nivel de vuelo controlado y sostenido, y propulsado por un motor de explosión, eléctrico o de reacción.

El diseño de los VANT tiene una amplia variedad de formas, tamaños, configuraciones y características.

Actualmente los UAV tienen numerosas aplicaciones en el ámbito civil y comercial (su uso se ha extendido en los últimos años), aunque en sus inicios eran utilizados principalmente con fines militares (tareas de reconocimiento, vigilancia y adquisición de objetivos).

Figura 1.1 UAV comercial.

Las principales ventajas de los vehículos aéreos no tripulados son las siguientes:

- No se necesita un piloto a bordo, lo que evita los posibles daños que pudiera sufrir y permite que las dimensiones del dron sean mucho más pequeñas.
- Uso en zonas aéreas peligrosas o de difícil acceso para los humanos.
- En el caso de los drones comerciales, su manejo no requiere de una alta cualificación.

En este trabajo, el dron utilizado es el descrito en el trabajo previo. Se trata de un quadrotor del que cuelga un gimbal de tres grados de libertad y en cuyo extremo se encuentra la cámara, cuya orientación queremos controlar para el seguimiento (*tracking*) del objetivo (*target*) deseado.

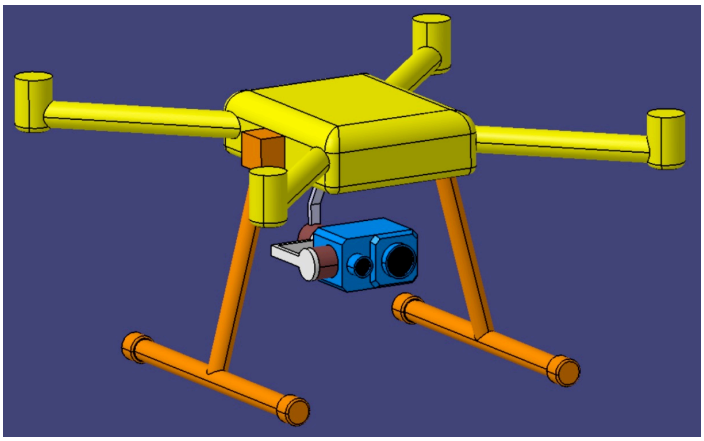

Figura 1.2 Diseño 3D final del dron.

1.2. Objetivos

Como se ha explicado en el resumen, el objetivo principal de este trabajo es el ensayo bajo simulación del sistema descrito para rastrear de forma autónoma un objetivo en movimiento. Para ello se hará uso del dron mostrado anteriormente, cuyo gimbal será controlado para conseguir el seguimiento de los objetivos con la cámara.

En un primer paso habrá que adquirir toda la información del proyecto del que partimos para ser capaces de hacer funcionar todo lo resuelto en el trabajo anterior. Tras identificar y comprender el funcionamiento de todos los componentes del sistema completo se pasa resolver el problema de este trabajo.

En primer lugar, hay que buscar algún algoritmo que extraiga los puntos característicos de una imagen para identificar con la cámara el objeto al que se quiere seguir.

Una vez se resuelva lo anterior, se pasa a encontrar una estrategia de control del gimbal para conseguir el seguimiento visual del objetivo.

1.3. Estructura del trabajo

Este trabajo se divide en los capítulos siguientes, además de la introducción:

- Capítulo 2: en el segundo capítulo se aborda el estado del arte del seguimiento de objetivos para ver cuáles son los métodos más empleados en este ámbito.
- Capítulo 3: este capítulo introduce y explica las herramientas que se van a usar en el proyecto, que son:
  - Gazebo.
  - ROS.
- Capítulo 4: en el cuarto capítulo se expone el algoritmo SIFT, escogido para la detección del objetivo del cual se quiere realizar el seguimiento.
- Capítulo 5: en el quinto capítulo se desarrolla la ley de control del gimbal necesaria para llevar a cabo el *tracking* de nuestro objetivo.
- Capítulo 6: en este capítulo se muestran los resultados de las distintas pruebas realizadas sobre la simulación de Gazebo.
- Capítulo 7: en el último capítulo se acaba con las conclusiones del proyecto y los posibles trabajos futuros.

2 ESTADO DEL ARTE

2.1. Introducción

En los últimos tiempos, se ha producido una explosión en el uso de la tecnología de seguimiento de objetos (*object tracking*) en aplicaciones no militares.

Los algoritmos de seguimiento de objetos se han convertido en una parte esencial de nuestra vida cotidiana. Por ejemplo, la navegación basada en el GPS es una herramienta cotidiana de la humanidad. En esta aplicación, un grupo de satélites artificiales en el espacio exterior continuamente localiza los vehículos que conducen las personas y los algoritmos de seguimiento de objetos en el GPS realizan la auto-localización y nos permiten disfrutar de una serie de servicios basados en la localización, como la búsqueda de lugares de interés y la planificación de rutas. Del mismo modo, el seguimiento de objetos se utiliza en una gran variedad de contextos, como la vigilancia del espacio aéreo, el seguimiento de vehículos, el seguimiento de submarinos y ballenas y la vigilancia inteligente por vídeo.

También se utilizan en la navegación de robots autónomos mediante láseres, cámaras estereoscópicas y otros sensores de proximidad.

El seguimiento de objetos/objetivos se refiere al problema de utilizar las mediciones de los sensores para determinar la ubicación, la trayectoria y las características de los objetos de interés. Un sensor puede ser cualquier dispositivo de medición, como un radar, un sonar, un ladar, una cámara, un sensor de infrarrojos, un micrófono, un ultrasonido o cualquier otro sensor que pueda utilizarse para recoger información sobre objetos en el entorno.

Los objetivos típicos del seguimiento de objetos son la determinación del número de objetos, su identidad y sus estados, como la posición, la velocidad y el tiempo de permanencia.

El seguimiento de objetos es una tarea importante dentro del campo de la visión por ordenador. La proliferación de ordenadores de gran potencia, la disponibilidad de cámaras de vídeo de alta calidad y bajo coste, y la creciente necesidad de automatizar el análisis de vídeo han generado un gran interés en los algoritmos de seguimiento de objetos. Hay tres pasos clave en el análisis de vídeo: la detección de objetos en movimiento interesantes, el seguimiento de dichos objetos de un fotograma a otro y el análisis del comportamiento de los objetos para predecir su trayectoria. Por tanto, el uso del seguimiento de objetos es pertinente en las tareas de:

- Reconocimiento basado en el movimiento.
- Vigilancia automatizada.
- Indexación de vídeos.
- Interacción persona-ordenador.
- Seguimiento del tráfico.
- Navegación de vehículos.

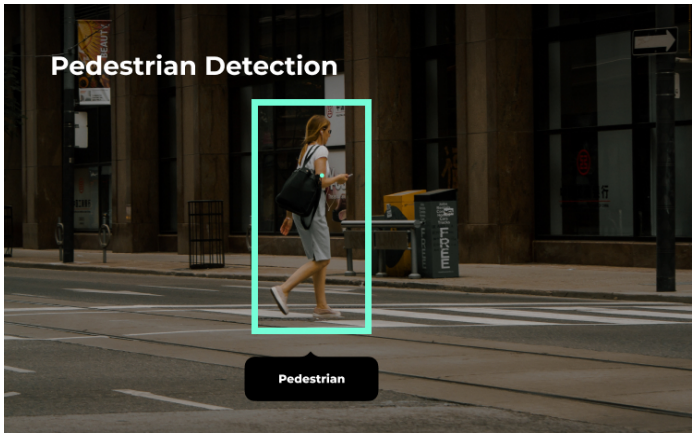

Figura 2.1 Detección de un peatón.

El seguimiento de objetos puede ser bastante complejo debido a:

- Pérdida de información causada por la proyección del mundo 3D en una imagen 2D,
- Ruido en las imágenes.
- El movimiento complejo de los objetos.
- La naturaleza no rígida o articulada de los objetos.
- Las oclusiones parciales y totales de los objetos, las formas complejas de los objetos.
- Cambios en la iluminación de la escena y requisitos de procesamiento en tiempo real.

Se puede simplificar el seguimiento imponiendo restricciones al movimiento y/o a la apariencia de los objetos. Por ejemplo, casi todos los algoritmos de seguimiento suponen que el movimiento del objeto es suave y sin cambios bruscos. Además, se puede restringir el movimiento del objeto para que sea de velocidad constante o de aceleración constante basándose en información a priori. El conocimiento previo sobre el número y el tamaño de los objetos, o el aspecto y la forma del objeto, también puede utilizarse para simplificar el problema.

2.2. Detección de objetos

Todos los métodos de seguimiento requieren un mecanismo de detección de objetos, ya sea en cada fotograma o cuando el objeto aparece por primera vez en el vídeo.

En la Tabla 2.1 se muestran varios métodos habituales de detección de objetos. Aunque la detección de objetos en sí misma es muy amplia, aquí se resumen los métodos más populares en el contexto del seguimiento de objetos para completar la información:

Tabla 2.1 Métodos para la detección de objetivos.

Categorías	Métodos
Point detectors	Moravec's detector [Moravec 1979], Harris detector [Harris and Stephens 1988], Scale Invariant Feature Transform [Lowe 2004], Affine Invariant Point Detector [Mikolajczyk and Schmid 2002].
Segmentation	Mean-shift [Comaniciu and Meer 1999], Graph-cut [Shi and Malik 2000], Active contours [Caselles et al. 1995].
Background Modeling	Mixture of Gaussians[Stauffer and Grimson 2000], Eigenbackground[Oliver et al. 2000], Wall flower [Toyama et al. 1999], Dynamic texture background [Monnet et al. 2003].
Supervised Classifiers	Support Vector Machines [Papageorgiou et al. 1998], Neural Networks [Rowley et al. 1998], Adaptive Boosting [Viola et al. 2003].

3 HERRAMIENTAS UTILIZADAS

En este apartado se van a exponer las herramientas utilizadas en el desarrollo de este proyecto. Para poder hacer uso de todos los archivos que se utilizaron en el trabajo previo es recomendable usar la misma versión de los programas empleados originalmente.

3.1. Gazebo

Gazebo es un simulador dinámico 3D con la capacidad de simular de forma precisa y eficiente poblaciones de robots en complejos ambientes interiores y exteriores. Aunque es similar a los motores de juegos, Gazebo ofrece una simulación física con un grado de fidelidad mucho más alto, un conjunto de sensores e interfaces tanto para usuarios como para programas. Sus características más relevantes son:

- Es un software libre compatible con ROS.
- Capacidad de desarrollar y simular modelos de robots propios (URDF).
- Posibilidad de crear escenarios ("mundos") de simulación.
- Contiene multitud de plugins para añadir sensores al modelo del robot y simularlos, como sensores de odometría, de fuerza, de contacto...

Figura 3.1 Logotipo de Gazebo.

Para no tener problemas de compatibilidad con el proyecto anterior, el sistema operativo elegido ha sido Ubuntu 18.04 LTS, en el que se ha instalado ROS Melodic, que por defecto incluye Gazebo 9.

3.2. ROS

ROS (del inglés Robot Operating System) es un *framework* de código abierto para el desarrollo de software para robots. Es una colección de liberías de software y herramientas que tienen como objetivo simplificar la tarea de crear un comportamiento robótico complejo y robusto en una amplia variedad de plataformas robóticas.

Figura 3.2 Logotipo de ROS.

Figura 3.3 Logotipo de ROS Melodic.

ROS se compone de cuatro partes principales:

- *Plumbing* (o *middleware* de comunicación): esta parte de ROS crea una red de programas (nodos) que se comunican entre sí enviando y recibiendo diferentes tipos de datos. Esta conexión entre los distintos nodos se visualiza mediante el denominado ROS Computation Graph.
- *Tools* (herramientas): las herramientas presentes en ROS ayudan a depurar y monitorear los datos que se están enviando y recibiendo por los nodos. Hay tanto herramientas gráficas (como Rviz o Gazebo) como herramientas de línea de comandos.
- *Capabilities* (capacidades): usando la característica de *plumbing*, se construyen muchos bloques de software para robots para dotarlos de navegación, percepción, manipulación, etc. Si alguien quiere implementar estas capacidades en su robot, puede simplemente reutilizar software de ROS sin tener que empezar de cero.
- *Ecosystem* (ecosistema): ROS está impulsado por miles de desarrolladores en todo el mundo que contribuyen y realizan el mantenimiento de miles de paquetes de ROS, tutoriales, Q&A, etc. ROS es uno de los *framework* más usados en la mayoría de universidades para investigación en robótica, además de las compañías robóticas para prototipar sus softwares.

Figura 3.4 Representación de la ecuación de ROS.

3.2.1 Elementos básicos de ROS

Nodos

Los nodos son archivos ejecutables dentro de un paquete de ROS. Se comunican entre ellos mediante los llamados *topics*. En este proyecto, los nodos suelen estar dentro de la carpeta *src* del paquete correspondiente.

Los nodos se encargan de realizar cálculos y operaciones, y publican o se suscriben a un *topic*. Se pueden programar tanto en C++ como en Python.

Mensajes

Los mensajes son dato con una estructura predeterminada (como *std_msgs* o *geometry_msgs*) o bien pueden ser personalizados por el usuario.

Los mensajes no se encuentran definidos en el interior de los códigos de los nodos que lo utilicen, sino que se crean en una carpeta distinta a la de los nodos, normalmente con el nombre *msg*.

Topics

Los *topics* son el medio de comunicación entre los distintos nodos. A través de ellos los nodos intercambian mensajes. Para ello, los nodos pueden tanto publicar como suscribirse al topic que deseen.

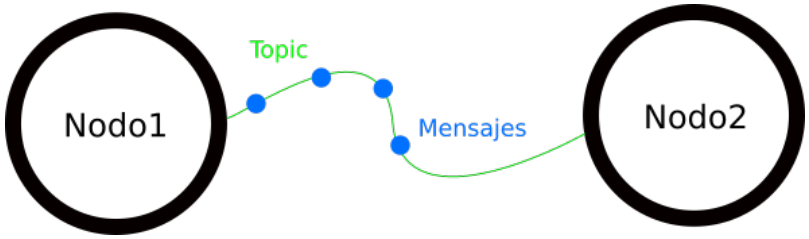

Figura 3.5 Imagen explicativa del intercambio de mensajes entre dos nodos.

Una vez se haya comprendido el funcionamiento de este proyecto se muestran resumidos en unas tablas los nodos y *topics* (con su mensaje asociado) de los que se ha hecho uso o que han sido creados. Dichas tablas se encuentran en el capítulo 5 (Tabla 5.1, Tabla 5.2).

3.3. Organización del proyecto

A continuación, se explican resumidamente los principales componentes en los que se divide el proyecto de ROS (o *workspace*) original del que se parte. Consta de tres paquetes:

- dron_description
  - meshes: carpeta que contiene los archivos del modelo 3D del dron.
  - urdf: incluye la descripción en formato URDF de los sólidos que componen el modelo del dron (Quadrotor + Gimbal + Cámara), las conexiones entre ellos y una serie de *plugins* asociados a los distintos cuerpos.

- dron_gazebo
  - worlds: contiene los elementos del entorno de la simulación.
  - plugins: conjunto de plugins que afectan al dron.
  - src: códigos para la conversión de datos de las IMU del quadrotor y de la cámara.
  - msg: archivos de mensajes de este paquete.
  - launch: archivo *.launch* (*launcher*) encargado de lanzar la simulación en Gazebo.

- ctrl_desacoplado

La función del tercer paquete es la de dotar al sistema de un control de vuelo para el quadrotor y del control de la orientación de la cámara.

- src: archivos que controlan tanto el gimbal como el quadrotor.
- msg: mensajes de este paquete.
- lauch: *launcher* que da comienzo al control del sistema en la simulación.
- config: archivo de configuración que permite obtener los valores de las coordenadas articulares del gimbal.

3.3.1 Lanzamiento de la simulación

Para iniciar la simulación, en primer lugar es necesario comprobar que en el archivo *ctrl_desacoplado.launch* que está en el paquete *ctrl_desacoplado* el nodo generador de referencias elegido sea el de *Referencias_Parabolico*, por lo que si este aparece comentado, habría que descomentarlo y en su lugar comentar el generador de *Referencias_Mando*.

La diferencia de estos dos nodos es que el de *Referencias_Parabolico* te permite dar las referencias de posición del quadrotor y la orientación de la cámara en el código de *Referencias_Parabolico.cpp*.

El nodo de *Referencias_Mando* otorga las referencias del dron a través de un mando o *gamepad*. En nuestro caso esta opción no se utilizará y nos quedaremos siempre con *Referencias_Parabolico.*

Desde la ventana de comandos, lanzamos la simulación con la siguiente instrucción:

```
ivan@ivan~ZenBook:~$ roslaunch dron_gazebo dron.launch
```

Esto abrirá la simulación en la aplicación de Gazebo.

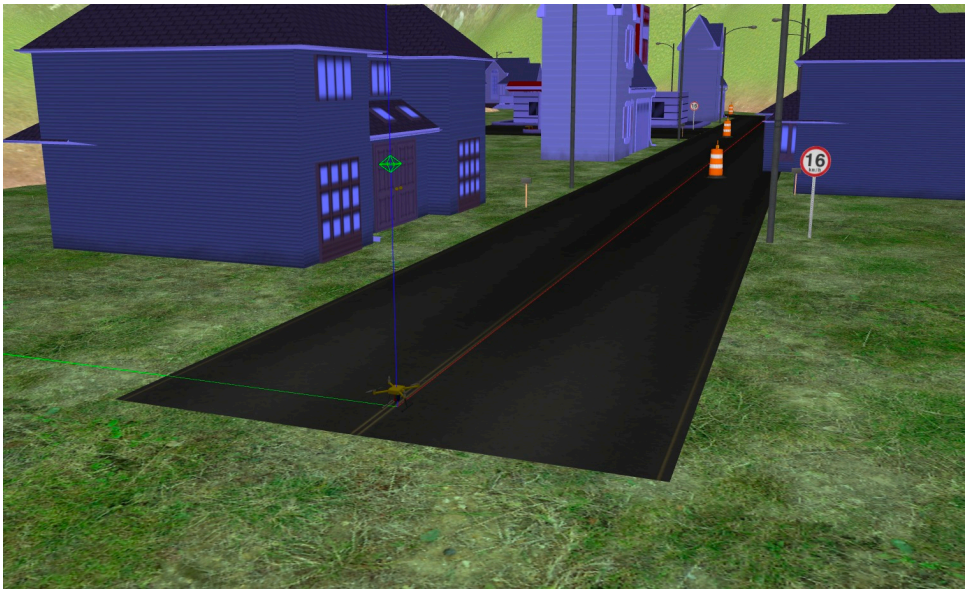

Figura 3.6 Simulación del "mundo" en Gazebo.

Tras esto, para aplicar el control de vuelo al dron y que este siga la trayectoria deseada descrita en el archivo *Referencias_Parabolico.cpp* habría que escribir lo siguiente en otra ventana de comandos:

```
ivan@ivan~ZenBook:~$ roslaunch ctrl_desacoplado ctrl_desacoplado.launch
```

4 DETECCIÓN DEL OBJETIVO

El primer paso tras comprender la función de todos los archivos del trabajo previo es la búsqueda de un algoritmo capaz de detectar visualmente el objetivo para posteriormente realizar su seguimiento con la cámara. En este capítulo se presenta el algoritmo finalmente elegido y su implementación en el proyecto.

4.1. SIFT

4.1.1 Introducción al algoritmo

SIFT (Scale Invariant Feature Transform) es un método que permite detectar puntos característicos en una imagen y luego describirlos mediante un histograma orientado de gradientes. Además, lo hace de forma que la localización y la descripción presenta una gran invarianza a la orientación, la posición y la escala. Cada punto característico queda, por lo tanto, definido mediante su vector de características de 128 elementos, y se obtiene la información de su posición en coordenadas de la imagen, la escala a la que se encontró y la orientación dominante de la región alrededor de dicho punto.

David Lowe ideó este nuevo algoritmo en 2004 en su artículo "Distinctive Image Features from Scale-Invariant Keypoints".

Hay principalmente cuatro pasos en el algoritmo SIFT:

1. Detección de extremos en el espacio de escala (mediante la diferencia de gaussianos, DoG)
2. Localización de los puntos clave o *keypoints*: usaron la expansión de la serie de Taylor del espacio de escala para obtener una ubicación más precisa de los extremos.
3. Asignación de orientación: con esto se consigue la invarianza a rotación.
4. Descriptor de puntos clave: se forma el descriptor de puntos clave, además de tomar varias medidas para lograr robustez ante cambios de iluminación, rotación, etc.

Figura 4.1 *Keypoints* detectados por algoritmo SIFT en dos imágenes.

Tras estos cuatro pasos, hay uno extra si lo que se quiere, además de obtener los *keypoints* de una imagen, es corresponderlos con otra imagen desde una perspectiva distinta. Ese sería el quinto y último paso:

5. Correspondencia de los *keypoints* de dos imágenes.

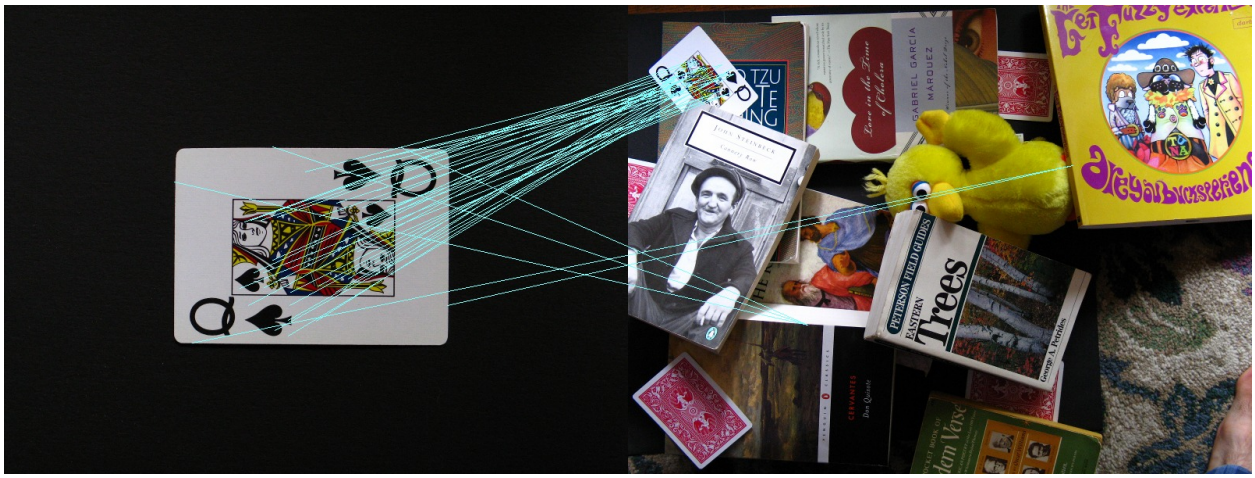

Figura 4.2 Correspondencia de los *keypoints* en ambas imágenes.

Las principales ventajas de SIFT son:

- Localidad: las características son locales, tan resistentes a la oclusión y al desorden (sin segmentación previa).
- Distinción: las características individuales se pueden combinar con una gran base de datos de objetos.
- Cantidad: se pueden generar muchas características incluso para objetos pequeños.
- Eficiencia: rendimiento cercano al tiempo real.
- Extensibilidad: se puede extender fácilmente a una amplia gama de diferentes tipos de funciones, cada una de las cuales agrega robustez.

Este algoritmo es el utilizado para la detección de objetivos en nuestro proyecto.

4.1.2 Integración en el proyecto

Al tratarse de un algoritmo bastante complejo, la programación de un código sería algo muy largo y laborioso, y dado que la implementación de este método no es el objetivo principal de este proyecto, se decidió por buscar algún repositorio ya desarrollado.

Después de tratar de implementar varios repositorios, que no funcionaban por problemas de compatibilidad con la versión de ROS o porque se requería un ordenador con tarjeta gráfica Nvidia (en mi caso la tarjeta gráfica es de AMD y fallaba), se encontró uno que, tras una serie de modificaciones (que se detallan a continuación), proporciona los resultados deseados.

El paquete de ROS que se ha utilizado es el de *image_pose_estimation* de Devitt Dmitry. Este paquete se encarga de obtener la posición en el "mundo" del objeto seleccionado. Está escrito en Python y utiliza OpenCV con el algoritmo SIFT.

Dicho paquete es independiente de todos los demás, por lo que ahora tendremos estos cuatro:

- dron_description
- dron_gazebo
- ctrl_desacoplado
- image_pose_estimation

En su interior encontramos dos carpetas:

- **launch:** contiene el archivo *image_pose_estimation.launch* encargado de lanzar el nodo correspondiente a este paquete. Se especifica la ruta de la carpeta con la imagen del objeto a detectar en el campo *image_path* de dicho archivo.
- **src:** aquí disponemos de los dos *scripts* en Python que se ocupan de detectar el objeto deseado y localizar su centro en la imagen de la cámara. Estos archivos son *image_pose_estimation_script.py* y *image_processing.py*.

image_pose_estimation_script.py

Este código se encarga de recoger los parámetros introducidos en el *launcher* (*image_pose_estimation.launch*) tales como el nombre de la cámara, el *path* de la imagen con el objeto a identificar y otros valores que se han dejado por defecto.

Desde este código se llama al *script* de *image_processing.py*, que devuelve una "clase" (*class*) de la cual nos interesan las variables de las coordenadas del centro del objetivo en la cámara y una variable *flag* que se explica a continuación.

Este *script* ha sido modificado para que se publique la posición del objetivo en la cámara a través de un *topic* (*/target_cam_pose*). Solo se publica a través de él cuando el valor de la variable *flag* es *True*.

image_processing.py

En este *script* se implementa el algoritmo SIFT con la ayuda de las funciones de OpenCV *SIFT_create()*, que construye un objeto SIFT y *sift.detectAndCompute()* que obtiene los *keypoints* y los descriptores en un solo paso.

También se hace uso de una función ya creada en el código encargada de realizar el *matching* de los puntos clave de la imagen del objetivo con el *frame* de la cámara en cada instante.

Se introdujo una variable *flag* cuya función es la de variable bandera, su valor por defecto es *False*, pero cuando el objetivo es identificado, es decir, cuando se supera un umbral (establecido en el parámetro *min_match_count* del archivo *.launch*) del número de correspondencias de *keypoints* de la imagen del objetivo con la imagen de la cámara, la variable cambia a *True*.

Problemas del paquete

Al tratar de compilar el proyecto (al hacer *catkin_make* en el *workspace*) aparecen una serie de errores tales como:

- fatal error: opencv2/nonfree/features2d.hpp: No such file or directory

Esto se debe a que no tenemos la librería *nonfree* de OpenCV, en la que se encuentran los algoritmos de SIFT y SURF, que se sabe que están patentados.

Una solución para obtener dicha librería es ejecutando las siguientes instrucciones en la línea de comandos:

```
ivan@ivan~ZenBook:~$ sudo add-apt-repository --yes ppa:jeff250/opencv
ivan@ivan~ZenBook:~$ sudo apt-get update
ivan@ivan~ZenBook:~$ sudo apt-get install libopencv-dev
ivan@ivan~ZenBook:~$ sudo apt-get install libopencv-nonfree-dev
```

4.1.3 Lanzamiento del nodo detector de objetivo

Para hacer uso de este paquete, una vez abierta la simulación (*launcher* del paquete *dron_gazebo*) y aplicado el control (*launcher* del paquete *ctrl_desacoplado*), se escribe lo siguiente en una ventana de comandos:

```
ivan@ivan~ZenBook:~$ roslaunch image_pose_estimation image_pose_estimation.launch
```

4.2. Detección de un objetivo estático

En este apartado se muestra un ejemplo del funcionamiento del paquete al lanzar el nodo, tomando una de las señales de tráfico que se encuentran en el "mundo" de Gazebo como objetivo a identificar.

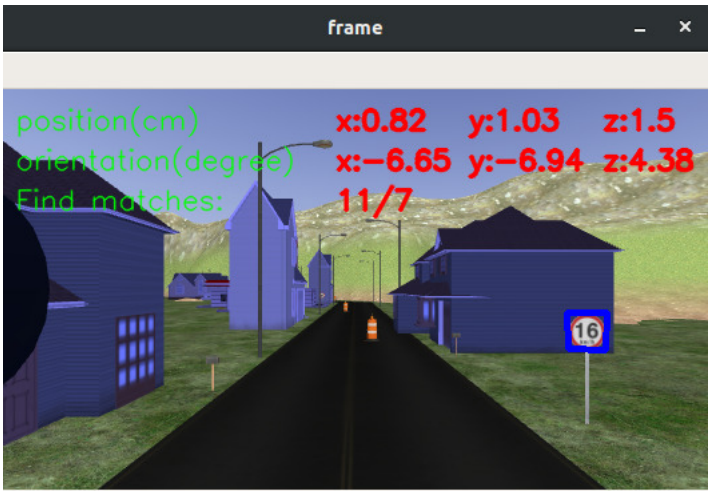

Figura 4.3 Objetivo identificado en la imagen de la cámara.

```
ivan@ivan-ZenBook:~$ rostopic echo /target_cam_pose
centre_x: 438.2684021
centre_y: 181.601013184
---
centre_x: 438.239715576
centre_y: 181.626037598
---
```

Figura 4.4 Coordenadas del centro del objetivo publicándose en el topic */target_cam_pose*.

Vemos cómo el algoritmo detecta correctamente el objeto (dibujando un cuadrado azul a su alrededor) y publica a través del topic */target_cam_pose* las coordenadas del centro del objetivo.

4.3. Detección de objetivo en movimiento

Tras comprobar el buen funcionamiento del algoritmo con un objetivo estático, ahora hay que comprobar la detección de un objeto que esté en movimiento.

4.3.1 Actor

Para ello, se ha introducido en la simulación un "actor". En Gazebo, un actor es un modelo animado. Los actores son como los modelos, pero con la diferencia que a los actores no se les aplica ninguna fuerza (como fuerzas de gravedad o de contacto).

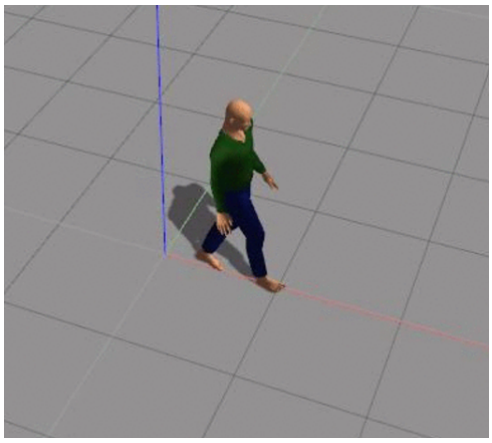

Figura 4.5 Actor en una simulación de Gazebo.

Un actor también puede describir una trayectoria definida directamente en SDF (dentro del archivo *dron.world*), por lo que es bastante sencillo de utilizar.

La trayectoria consiste en especificar una serie de posiciones que el actor tiene que alcanzar en tiempos determinados. Gazebo se encarga de interpolar el movimiento entre ellos para que la animación sea fluida.

La trayectoria elegida en nuestro caso ha sido una línea recta, de manera que nuestro actor cruza la carretera de un lado hacia el otro en un bucle.

Para ello, en el archivo *dron.world* se incluye el actor, especificando su apariencia (*skin*), la animación a realizar (*animation*) y la trayectoria deseada (*trajectory*). La apariencia y la animación elegida es la misma en ambos casos, *walk.dae* (ya instalado con Gazebo). El código correspondiente se encuentra en el Apéndice A.

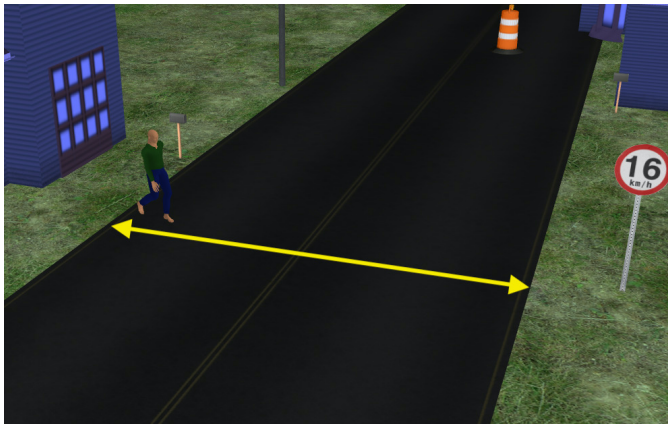

Figura 4.6 Trayectoria seguida por el actor en la simulación.

Modificaciones

Como el objetivo en este caso está en movimiento, se ha decidido aportar varias imágenes del objetivo para ser detectado. Al igual que en la detección de un objetivo estático, en el *launcher* del paquete se especifica la ruta de la carpeta que incluye las imágenes del objetivo. Esto se ha visto necesario ya que el actor no simplemente cambia su posición, sino que al estar animado mueve brazos y piernas y cambia su orientación.

Para ello, en el *script image_processing.py* se aplica el algoritmo SIFT y se realiza el *matching* de la imagen de la cámara con todas las imágenes que haya en la carpeta proporcionada, de manera que, si no hay correspondencia con una de ellas, se pueda comprobar la correspondencia con otra imagen distinta.

Problemas

Tras realizar numerosas pruebas se comprobó que el algoritmo SIFT no detectaba del todo bien a dicho actor, ya que el personaje no tiene muchas peculiaridades para ser detectado correctamente. Esto hacía necesario bajar el umbral de correspondencia de los *keypoints*, lo que llevaba a la detección errónea del objetivo.

4.3.2 Modelos de Gazebo

Aunque el uso de un actor para tener un objeto en movimiento a priori era la mejor opción, hubo que buscar otras alternativas, por lo que se decidió utilizar un modelo de Gazebo con características más notables.

El primer paso es introducir el modelo deseado en el archivo *dron.world* para cargarlo con la simulación. En dicho archivo le damos su posición inicial en el "mundo".

Para hacer que nuestro modelo estuviera en movimiento se programó un *script* sencillo (ver en Apéndice A) donde se obtiene la posición actual del modelo mediante la llamada al servicio *GetModelState*, y tras ello publicar a través del topic */gazebo/set_model_state* la nueva posición que queremos que adquiera el modelo.

Al servicio *GetModelState* es necesario pasarle dos parámetros: el nombre del modelo de Gazebo a utilizar (*model_name*) y el nombre de la entidad relativa (*relative_entity_name*).

El primero es directamente el nombre del modelo que aparece en Gazebo, y el segundo es el nombre que se obtiene del campo *body_names* que proporciona el servicio *GetModelProperties* (a este servicio solamente es neserario pasarle el nombre del modelo), que suele ser "link".

La trayectoria seguida por el modelo elegido es una línea recta que atraviesa la carretera, de la misma manera que lo hacía el actor. Para ello se incrementa la coordenada Y del modelo poco a poco hasta alcanzar un límite, donde se detiene por unos segundos. Tras ello, empieza a decrementar su valor hasta llegar al otro límite establecido, donde se vuelve a detener por unos instantes. Así simulamos un bucle en el que el modelo va hacia adelante y hacia atrás.

El modelo elegido para realizar las pruebas de seguimiento ha sido el de una camioneta (*pickup*), ya que, como se puede ver en la imagen mostrada a continuación, el número de *matches* con este modelo es bastante alto, lo que hace que el algoritmo tenga menos error y detecte correctamente la camioneta.

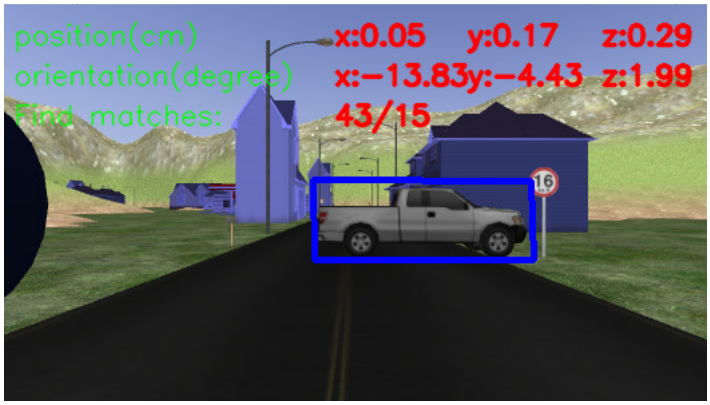

Figura 4.7 Detección del modelo de *pickup*.

También se ha probado con otros modelos como el de *person_walking*:

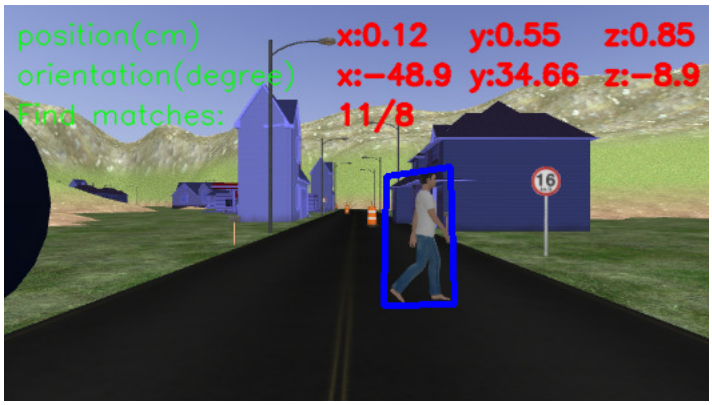

Figura 4.8 Detección del modelo de *person_walking*.

Como se observa en este caso, el número de *matches* es mucho más bajo, por lo que hay mucho más error que con el modelo de la camioneta.

4.3.3 Lanzamiento del nodo para el movimiento de la camioneta

Al ejecutar esta instrucción por línea de comandos la camioneta empezará a moverse:

```
ivan@ivan~ZenBook:~$ python ~/catkin_ws/src/dron_gazebo/scripts/mueve_pickup.py
```

5 SEGUIMIENTO DEL OBJETIVO

Este capítulo está dedicado a la parte del control del gimbal para realizar la tarea del seguimiento del objetivo identificado anteriormente por el algoritmo SIFT.

En un primer paso se probó un control básico, proporcional al error de las coordenadas del objeto con el centro de la cámara, para comprobar que todo funcionaba correctamente.

Finalmente, para el control del gimbal se ha tomado el paper "Gimbal Control for Vision-based Target Tracking" ([2]) como referencia para resolver el problema de *tracking*.

5.1. Ley de control del gimbal

Este documento soluciona el problema del control de un gimbal de tres ejes que lleva una cámara, de la cual se utilizan las imágenes para realimentar el control. La finalidad es la de mantener el objetivo de interés en el centro de la imagen de la cámara. Se procede a la explicación de la solución propuesta por este paper.

En primer lugar, antes de comenzar con la explicación hay que decir que la matriz que genera el operador S (expresión 2 del documento) es errónea:

$$S(a) = \begin{bmatrix} 0 & a_3 & -a_2 \\ -a_3 & 0 & a_1 \\ a_2 & -a_1 & 0 \end{bmatrix} \tag{5.1}$$

Como bien se dice en la línea debajo de dicha expresión en el paper, $S(a)b = a \times b$, siendo × el producto cruzado (o producto vectorial). Se comprobó que el producto escalar $S(a) \cdot b$ no daba el mismo resultado que el producto cruzado de a y b, siendo $S(a)b = -(a \times b)$.

$$S(a)b = \begin{bmatrix} a_3 \cdot b_2 - a_2 \cdot b_3 \\ -a_3 \cdot b_1 + a_1 \cdot b_3 \\ a_2 \cdot b1 - a_1 \cdot b_2 \end{bmatrix} \qquad a \times b = \begin{bmatrix} -a_3 \cdot b_2 + a_2 \cdot b_3 \\ a_3 \cdot b_1 - a_1 \cdot b_3 \\ -a_2 \cdot b1 + a_1 \cdot b_2 \end{bmatrix} \tag{5.2}$$

Esto se ha corroborado con una búsqueda en Internet. En la teoría helicoidal, un uso de este operador, representado por el acento circunflejo (â en lugar de S(a)), es el de representar el producto vectorial. Se concluye que la forma correcta de dicha matriz expresada en (5.1) es:

$$S(a) = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{5.3}$$

El modelo del Gimbal describe cómo varía la matriz de rotación (R_C) que me lleva de la cámara a los ejes mundo para una velocidad angular del gimbal (omega) determinada.

$$\dot{R_C} = R_C S(\omega) \tag{5.4}$$

El control se plantea fijando el valor que debería tener esta omega en cada instante de tiempo para conseguir el objetivo propuesto, que R_C converja a R_C^* (la orientación deseada).

$$\omega = -kS^{-1}(R_e - R_e{}') \tag{5.5}$$

Esta expresión da como resultado la omega de control, suponiendo que el error R_e en orientación entre la cámara y el objetivo es conocido. La ecuación (5.5) hay que interpretarla del siguiente modo. Cuando aparece el operador S^{-1} significa que se trata de la operación inversa a la expresada en (5.3):

Es decir, R_e-R_e' (R_e menos su traspuesta) es una matriz antisimétrica que siempre tendrá la misma estructura que la matriz (5.3). Por lo tanto, $S^{-1}(R_e - R_e') = [e1\ e2\ e3]'$, es decir, un vector columna.

Luego según (5.5), $\omega = -kS^{-1}(R_e - R_e')$, donde e1, e2, e3 son conocidos si se tiene Re, y k es en principio cualquier valor positivo. Cuanto mayor sea k más agresivo será el control.

Lo que sigue es como calcular R_e, que es el *Lemma 1*. La expresión (16) del documento dice cómo sacar R_e como función de q y a:

$$R_e = \left[-\frac{S(q)^2 a}{||S(q)^2 a||} \quad \frac{S(q)a}{||S(q)a||} \quad \frac{q}{||q||}\right] \tag{5.6}$$

Esta expresión también ha sido corregida, ya que el denominador de la primera columna tiene una errata. La fórmula que aparece en el documento tiene como denominador $||S(q)a||$, y como se dice después que todas las columnas de R_e deben tener norma 1, se ha deducido que faltaba un cuadrado al operador *S*.

De esta ecuación podemos decir lo siguiente:

- **q** es el vector que une el origen del marco del target {T} con el origen de la cámara {C}. Es la dirección en que queremos que mire la cámara. El problema es que q no es conocida porque no tenemos las coordenadas 3D del target (p_T), solo conocemos su proyección en el marco de la cámara, que es **y**. Como se dice en el apartado B, el eje de la cámara y el target están alineados (nuestro objetivo), cuando el eje Z de la cámara (en nuestro caso es el eje X), que se supone que apunta según el eje óptico de ésta, y **q** están alineados.
- **a** es la medida de la aceleración que mide el Gimbal. Supone que hay poca aceleración en el movimiento y que la acelaración que mide la IMU del Gimbal aproximadamente se debe solo a la gravedad.

Suponiendo que **a** se obtiene de la IMU, solo queda calcular **q**, que sale de la expresión (17) del paper. Esta expresión depende solo de **y** (posición del *target* en la imagen de la cámara).

$$\frac{q}{||q||} = \frac{A^{-1}\begin{bmatrix} y \\ 1 \end{bmatrix}}{||A^{-1}\begin{bmatrix} y \\ 1 \end{bmatrix}||} \tag{5.6}$$

Lo que se obtiene en esta ecuación es **q**/||**q**||, pero eso no es problema porque las columnas de R_e tienen norma 1. Solo es necesario conocer su dirección, no su magnitud. Por tanto, el valor obtenido en (5.6) será el utilizado como **q** en el cálculo de R_e.

El vector **y** se obtiene del cálculo del centro del objetivo en la cámara, realizado por el algoritmo SIFT. No hay más que suscribirse al *topic /target_cam_pose* para obtener su valor.

A es la matriz de parámetros intrínsecos de la cámara. Esto se han obtenido fácilmente viendo su valor en el parámetro K del *topic /dron/camara/camera_info*:

$$A = \begin{bmatrix} 315.8158 & 0 & 265.5 \\ 0 & 315.8158 & 150.5 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.7}$$

Con esto ya se tiene todo lo necesario para obtener el valor de **q**, el cual introducimos en el cálculo de R_e.

No había ninguna IMU asociada al gimbal de la que pudieramos extraer los datos del acelerómetro para el vector **a**, por lo que hubo que buscar algún *plugin* para obtener este dato.

En la página web de Gazebo se encontró un *plugin* que simula un sensor IMU llamado GazeboRosImu. Este *plugin* publica la información a través de mensajes de tipo *sensor_msgs/Imu*. La estructura de este mensaje incluye un campo llamado *linear_acceleration*, que proporciona la aceleración lineal en los tres ejes.

Pero resulta que este sensor no tiene en cuenta la fuerza gravitatoria, por lo que esta es omitida. Como solución se encontró un *plugin* que sí incluía dicha fuerza, como en las IMUs reales. Se trata de un *plugin* perteneciente al paquete *hector_gazebo_plugins*, llamado de la misma forma (GazeboRosImu). Dicho *plugin* también utiliza mensajes del tipo *sensor_msgs/Imu*.

Para hacer uso del paquete *hector_gazebo_plugins* primero hay que instalar las dependencias:

```
ivan@ivan~ZenBook:~$ sudo apt-get install ros-melodic-hector-gazebo-plugins
```

Después pasamos a clonar el paquete en el *workspace* del proyecto:

```
ivan@ivan~ZenBook:~$ cd catkin_ws/src
ivan@ivan~ZenBook:~$ git clone
https://gitlab.iri.upc.edu/labrobotica/ros/sensors/imu/iri_imu_gazebo.git
```

Finalmente, para incluir este sensor en el modelo del dron es necesario incluir lo siguiente en el archivo *urdf* del paquete *dron_description*:

Código 5.1 Plugin *libhector_gazebo_ros_imu.so*.

```
<gazebo>
        <plugin name="accel_imu_sim" filename="libhector_gazebo_ros_imu.so">
            <updateRate>500.0</updateRate>
            <bodyName>Gimbal2</bodyName>
            <frameId>world</frameId>
            <topicName>/dron/imu_gimbal</topicName>
            <accelOffset>0 0 0</accelOffset>
            <accelDrift>0 0 0</accelDrift>
            <accelGaussianNoise>0 0 0</accelGaussianNoise>
            <rateOffset>0 0 0</rateOffset>
            <rateDrift>0 0 0</rateDrift>
            <rateGaussianNoise>0 0 0</rateGaussianNoise>
            <yawOffset>0</yawOffset>
            <yawDrift>0</yawDrift>
            <yawGaussianNoise>0</yawGaussianNoise>
        </plugin>
</gazebo>
```

Gracias a este *plugin* ya disponemos de las medidas de **a** del gimbal (el *plugin* se asocia al "Gimbal2"), que son publicadas continuamente a través del topic */dron/imu_gimbal*.

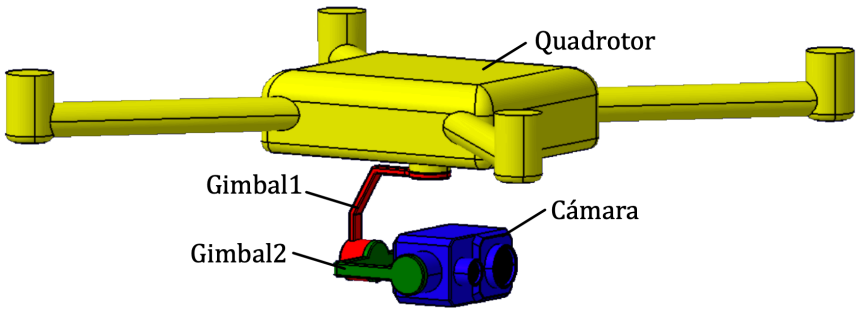

Figura 5.1 Nomenclatura para los sólidos del modelo del dron.

Ahora se dispone de todos los datos para hallar la matriz R_e.

Restando dicha matriz a su traspuesta se obtiene otra matriz de la forma:

$$R_e - R_e{}' = \begin{bmatrix} 0 & -e_3 & e_2 \\ e_3 & 0 & -e_1 \\ -e_2 & e_1 & 0 \end{bmatrix} \tag{5.8}$$

Por tanto, como se explicó antes, $S^{-1}(R_e - R_e{}') = [e1 \;\; e2 \;\; e3]'$.

Finalmente, solo quedaría ajustar la constante k para completar la ley de control (5.5).

Faltan por añadir unas modificaciones a realizar a la hora de implementar el control, ya que los ejes que se utilizan en el documento no coinciden con los ejes de nuestro dron:

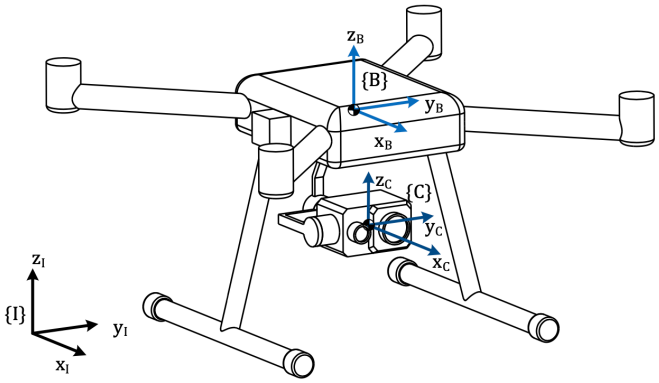

Figura 5.2 Sistemas de referencia principales en la posición de reposo.

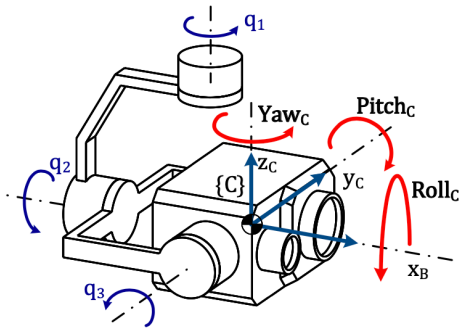

Figura 5.3 Ángulos de rotación de la cámara y coordenadas articulares del gimbal.

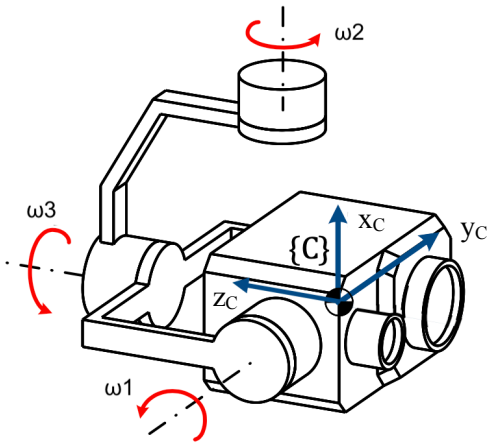

Figura 5.4 Sistema de referencia de la cámara según el paper.

Para la obtención del vector de aceleraciones del gimbal, habría que pasar las medidas que nos proporciona la IMU según nuestro sistema de coordenadas al sistema de referencia según el paper. Por lo que:

$$a = \begin{bmatrix} accel\_z \\ accel\_y \\ -accel\_x \end{bmatrix} \tag{5.9}$$

calculo_wref.py

Todos estos cálculos han sido plasmados en un nuevo *script* en Python, llamado *calculo_wref.py* (el código completo se encuentra en el Apéndice A).

Este *script* crea un nodo más del proyecto, y se encuentra dentro del paquete *ctrl_desacoplado*, en el interior de una carpeta llamada *scripts*.

En su interior se realiza la suscripción a dos topics para realizar los cálculos:

- */dron/imu_gimbal*: para obtener las aceleraciones de la IMU del gimbal, necesarias para disponer del vector **a**.
- */target_cam_pose*: del cual se adquieren las coordenadas del centro del objetivo en la imagen de la cámara y así formar el vector **y**.

Tras ello, se ha creado una función que realiza todos los cálculos para finalmente obtener la ley de control del gimbal, es decir, las velocidades angulares de referencia.

El lenguaje utilizado ha sido Python, ya que las distintas operaciones con matrices, como la inversión, la trasposición y la multiplicación, se realizan de manera muy sencilla con la librería *Numpy*, con la ayuda de las funciones *numpy.linalg.inv()*, *numpy.transpose()* y *numpy.dot()* respectivamente. Además, esta librería permite también obtener de manera rápida cálculos como la norma de un vector (*numpy.norm()*).

Para la publicación de las velocidades angulares de referencia (*wref*), se ha creado un mensaje con un único campo en su interior, un vector flotante de tres componentes (*float64[3]*). Este mensaje ha recibido el nombre de *calc_wref_msg.msg*.

5.1.1 Lanzamiento del nodo calculo_wref

Para lanzar este nodo se realizan los mismos pasos que en el apartado 4.2.1, y tras ello se escribe la siguiente instrucción por línea de comandos:

```
ivan@ivan~ZenBook:~$ python ~/catkin_ws/src/ctrl_desacoplado/scripts/calculo_wref.py
```

5.2. Control en velocidad del gimbal

El control del gimbal (*Control_Gimbal.cpp* dentro de la carpeta *src* del paquete *ctrl_desacoplado*) es el encargado de determinar los pares que deben ejercer los motores para llevar a las coordenadas articulares q1, q2, q3 a los valores que se han calculado previamente que permiten orientar la cámara de acuerdo con las referencias de orientación absoluta recibidas (Roll, Pitch, Yaw).

En el trabajo realizado por Fernando Borrego ([3]) se obtuvieron las funciones de transferencia que modelan el sistema del gimbal:

$$G_1(s) = \frac{1}{s(6.3 \cdot 10^{-4}s + 1.35 \cdot 10^{-5})} \qquad G_2(s) = \frac{1}{s(5.2 \cdot 10^{-4}s + 1.35 \cdot 10^{-5})}$$

$$G_3(s) = \frac{1}{s(1.8 \cdot 10^{-4}s + 1.35 \cdot 10^{-5})} \tag{5.10}$$

Estas funciones de transferencia tienen como salida la posición de las coordenadas articulares del gimbal.

Como en nuestro caso el control va a ser en velocidades angulares, el sistema del gimbal se modelaría de la siguiente manera:

$$G_1(s) = \frac{1}{6.3 \cdot 10^{-4}s + 1.35 \cdot 10^{-5}} \qquad G_2(s) = \frac{1}{5.2 \cdot 10^{-4}s + 1.35 \cdot 10^{-5}}$$

$$G_3(s) = \frac{1}{1.8 \cdot 10^{-4}s + 1.35 \cdot 10^{-5}} \tag{5.11}$$

Simplemente habría que eliminar el integrador de las funciones de transferencia para obtener a la salida velocidades angulares de las articulaciones del gimbal.

El controlador hallado en este caso ha sido calculado por cancelación de dinámica, quedando un control proporcional e integral (PI) para cada articulación:

$$C_1(s) = \frac{k_{C1}}{s}\ \frac{1}{G_1(s)} = \frac{k_{C1}}{s}\ (6.3 \cdot 10^{-4}s + 1.35 \cdot 10^{-5}) = k_{P1}\ \frac{46.667s + 1}{46.667s}$$

$$C_2(s) = \frac{k_{C2}}{s}\ \frac{1}{G_2(s)} = \frac{k_{C2}}{s}\ (5.2 \cdot 10^{-4}s + 1.35 \cdot 10^{-5}) = k_{P2}\ \frac{38.52s + 1}{38.52s}$$

$$C_3(s) = \frac{k_{C3}}{s}\ \frac{1}{G_3(s)} = \frac{k_{C3}}{s}\ (1.8 \cdot 10^{-4}s\ +\ 1.35 \cdot 10^{-5}) = k_{P3}\ \frac{13.333s + 1}{13.333s} \qquad (5.12)$$

Siendo $k_{P1} = 0.1$, $k_{P2} = 0.09$, $k_{P3} = 0.03$.

Los valores de estas constantes proporcionales han sido obtenidos experimentalmente probando en Matlab (Simulink) el funcionamiento del sistema del gimbal.

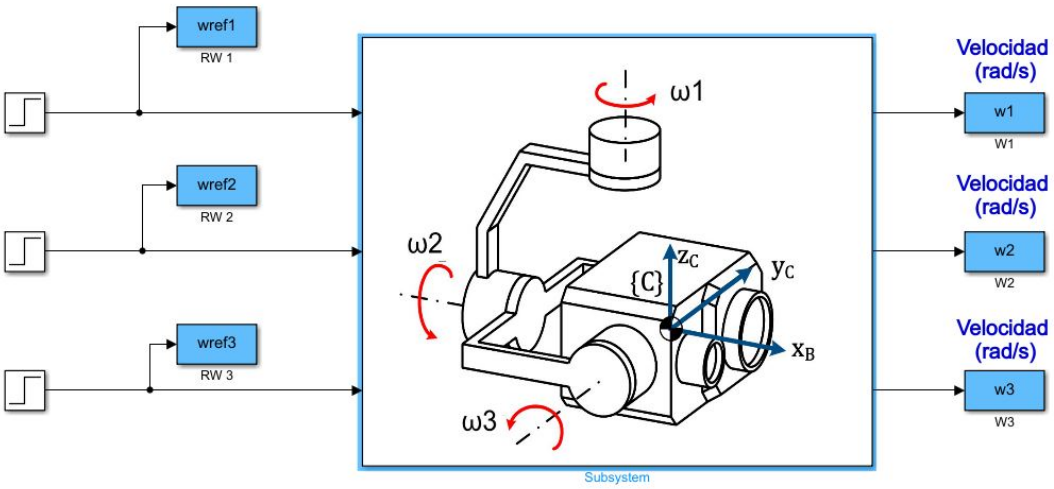

Figura 5.5 Esquema de bloques en Simulink.

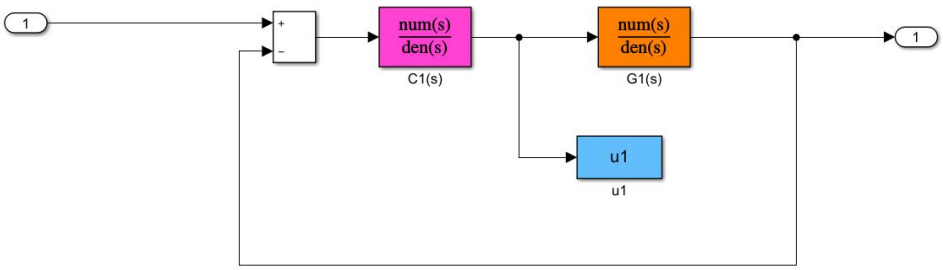

Figura 5.6 Detalle del interior del subsistema.

Dentro del subsistema creado encontramos tres estructuras de control como la de la Figura 5.6, una por cada articulación del gimbal.

Se probó el seguimiento de escalones de referencia para corroborar su comportamiento y obtener el tiempo de subida deseado. Particularmente, para escalones de referencia de 0.5 rad/s, se obtuvo lo siguiente:

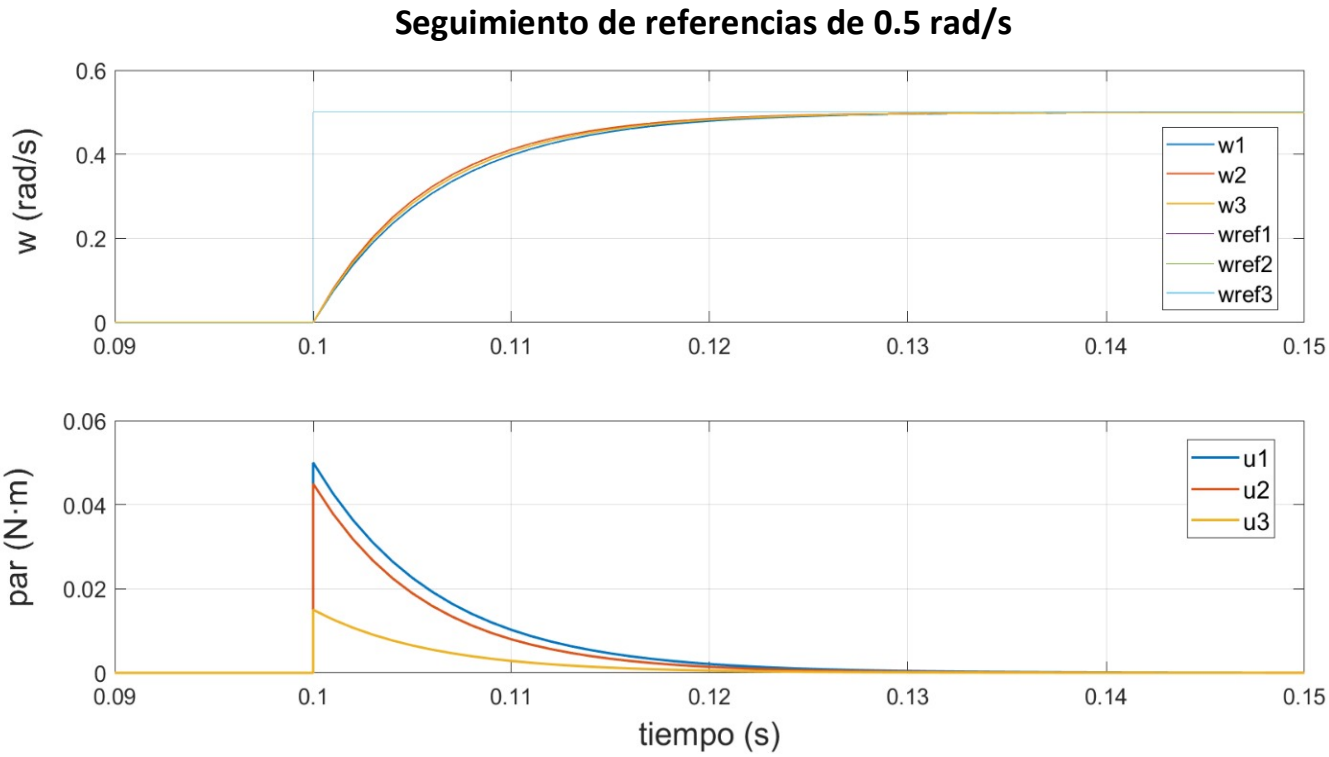

Figura 5.7 Seguimiento de referencias en velocidad angular del gimbal.

En esta gráfica se observa que los tiempos de subida (prácticamente iguales para los 3 motores) rondan los 0.015 segundos.

5.2.1 Integración en el proyecto

El siguiente paso sería implementar este control en nuestro proyecto. Para ello, hay que modificar el controlador del código *Control_Gimbal.cpp*, de forma que las articulaciones del gimbal giren según las velocidades angulares de referencias calculadas por la ley de control del documento anterior.

Modificamos los valores del controlador PID del código de *Control_Gimbal.cpp* para obtener el controlador PI calculado en el último apartado, por lo que quedaría de la siguiente forma:

Código 5.2 Trozo de código (parámetros del controlador) de *Control_Gimbal.cpp*.

```
…
/* Parámetros q0, q1 y q2 (aquí: a, b, c), para la implementación del PID */
/* Parámetros de un PID discretizado por Euler II:
   q0 = kp * (1 + T*(1/Ti) + Td/T) = kp * (1 + T*I + D/T)    --> a
   q1 = kp * (- 1 - 2*Td/T) = kp * (- 1 - 2*D/T)             --> b
   q2 = kp * Td/T = kp * D/T                                 --> c      */
```

```
double P2[3] = {        0.1,        0.09,       0.03};
double I2[3] =  { 0.02143,    0.02596,      0.075};
double D2[3] ={         0.0,          0.0,        0.0};
…
```

En el código, la variable I es la inversa de la constante integral.

La parte en la que se aplican dichos controles también cambia, pero solamente en el modo de funcionamiento normal del dron (en el arranque y aterrizaje del dron se dejan los valores originales del control en posición). Se puede ver la implementación en el Apéndice A, en el código de *Control_Gimbal.cpp*.

El control del motor 2 del gimbal (aquel que gira alrededor del eje X de la cámara) no se ha utilizado en este trabajo, ya que para el seguimiento del objetivo que sigue una trayectoria rectilínea no se vio necesario, únicamente controlando el giro alrededor de los ejes Y (Pitch) y Z (Yaw). Por tanto, en esta articulación se mantiene el control que venía originalmente, el de posición.

Como el control está directamente en el archivo *Control_Gimbal.cpp*, ya se ejecuta cuando se inicia el *launcher* del paquete *ctrl_desacoplado*, pero se ha programado de tal manera que el control en velocidad no se inicia hasta que no se lanza el nodo *calculo_wref*.

En este punto ya tenemos todo lo necesario para realizar los experimentos de *tracking* de objetivos. Para una mejor comprensión de los nodos sobre los que se ha trabajado y los topics que establecen la comunicación entre ellos se han creado unas tablas esquemáticas:

Tabla 1.1 Nodos.

Nodo	Código que lo inicia	Paquete al que pertenece
/control/Control_Gimbal	Control_Gimbal.cpp	ctrl_desacoplado
/image_pose_estimation_node	image_pose_estimation.py	image_pose_estimation
/pickup_movement	mueve_pickup.py	dron_gazebo
/calculo_wref	calculo_wref.py	ctrl_desacoplado

Tabla 2.2 *Topics* y nodos que los utilizan.

Topic	Mensaje	Paquete al que pertenece	Nodo que lo utiliza
/dron/imu_camara	Odometry.msg	nav_msgs	/control/Control_Gimbal
/target_cam_pose	target_cam_pose_msg.msg	image_pose_estimation	/calculo_wref
/dron/gazebo/setmodelstate	ModelState.msg	gazebo_msgs	/pickup_movement
/parametros/wref	calc_wref_msg.msg	ctrl_desacoplado	/control/Control_Gimbal
/dron/imu_gimbal	Imu.msg	sensor_msgs	/calculo_wref
/dron/camara/image_raw	Image.msg	sensor_msgs	/image_pose_estimation
/dron/camara/camera_info	CameraInfo.msg	sensor_msgs	/image_pose_estimation

Una vez se lanzan todos los nodos del proyecto, la estructura creada se puede observar mediante el esquema obtenido con *rqt_graph*, donde se ven los nodos del proyecto y la conexión entre ellos a través de los *topics*. La instrucción a escribir por línea de comandos es simple:

```
ivan@ivan~ZenBook:~$ rqt_graph
```

La figura resultante se muestra en la siguiente página:

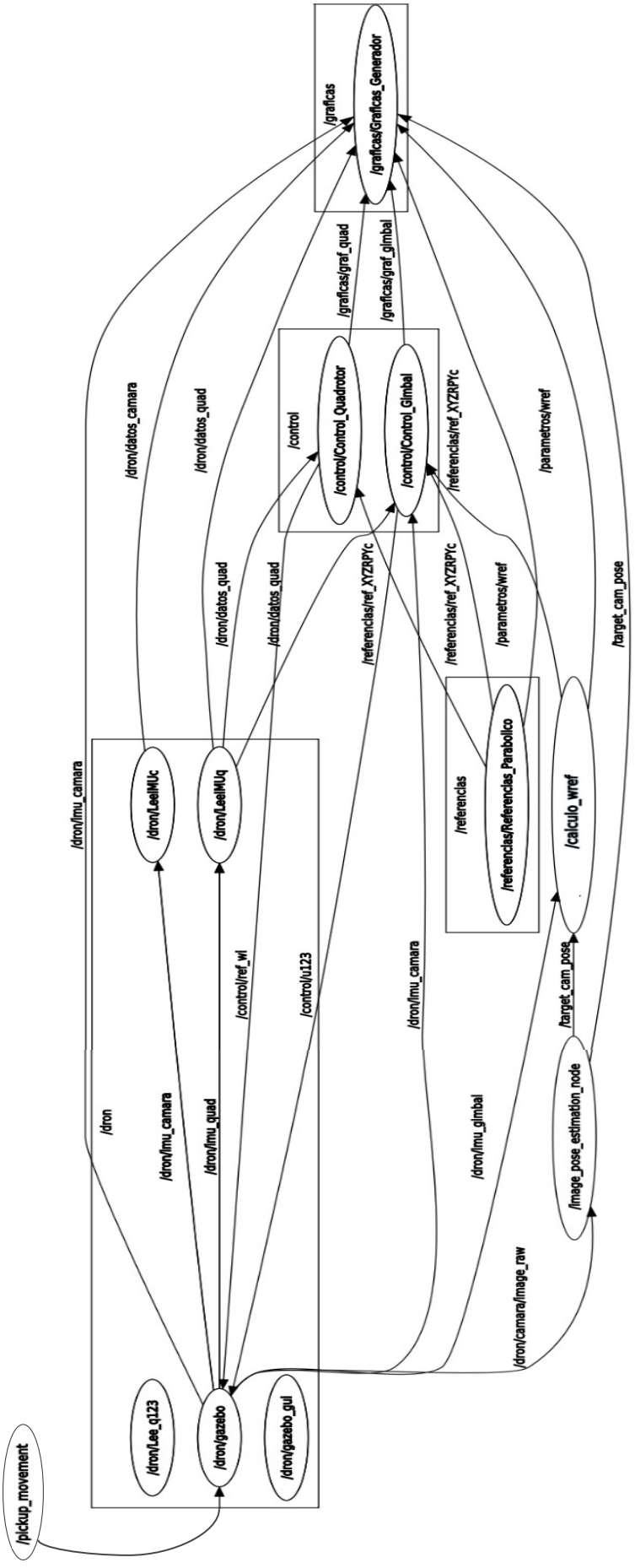

Figura 5.8 Esquema de la conexión entre nodos obtenido mediante *rqt_graph*.

6 ANÁLISIS DE RESULTADOS

En este capítulo se muestran los resultados de varias pruebas realizadas siguiendo un objetivo con la cámara del dron en distintas situaciones.

Las simulaciones en las que se basan las distintas pruebas han sido grabadas y se pueden ver a través de este enlace de GitHub: https://github.com/ivanf8/target-tracking/blob/main/README.md.

Haciendo click en las imágenes que aparecen, se abre el enlace del vídeo correspondiente en YouTube.

6.1. "Seguimiento" de un objetivo estático

Se ha realizado esta prueba con dos objetivos distintos, una señal de tráfico y una ambulancia.

Señal de tráfico

Antes de apuntar al centro del objetivo, la señal se encuentra en la parte superior derecha de la imagen de la cámara.

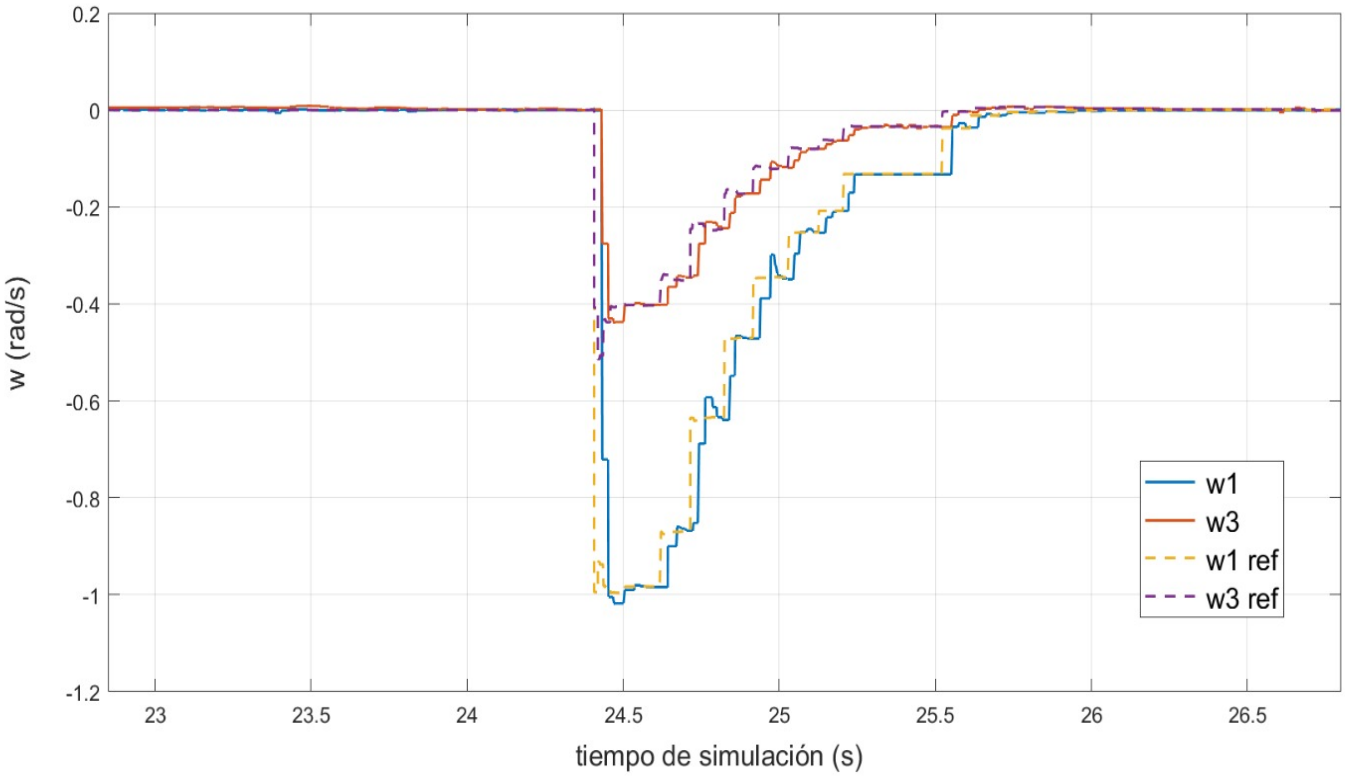

Figura 6.1 Seguimiento de las velocidades angulares de referencia para el gimbal (prueba 1).

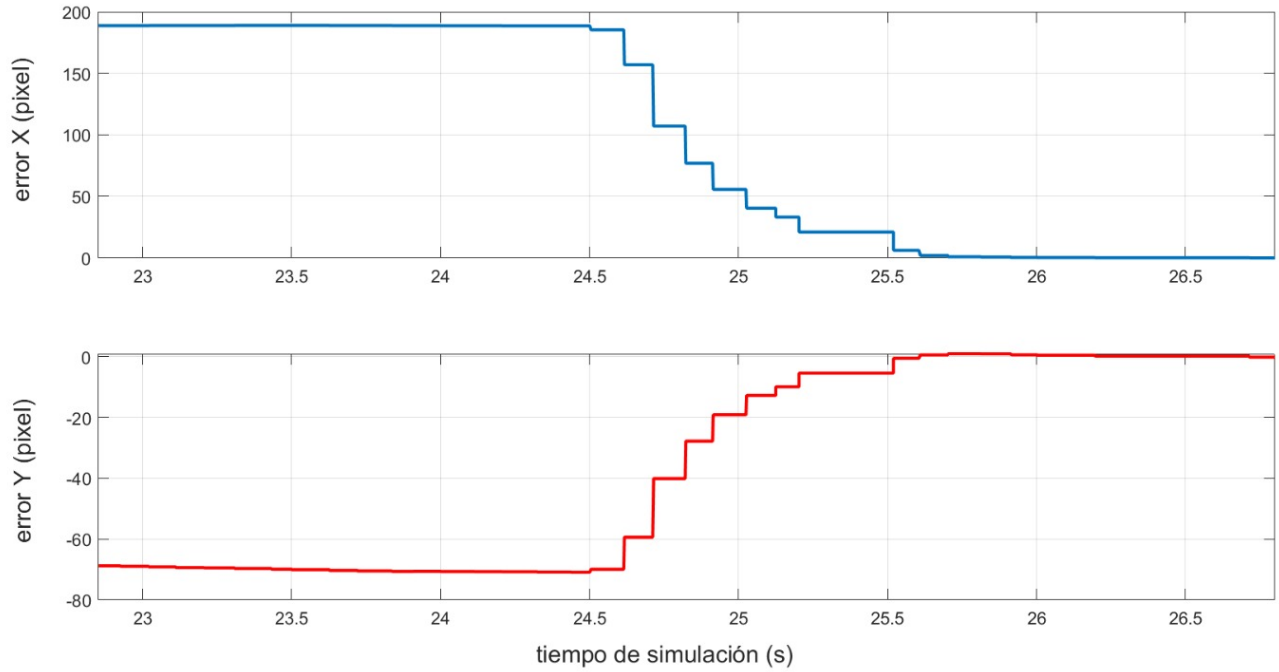

Figura 6.2 Error respecto al centro de la cámara (prueba 1).

Ambulancia

Al principio situamos la cámara de tal manera que la ambulancia quede en la parte inferior izquierda de la imagen.

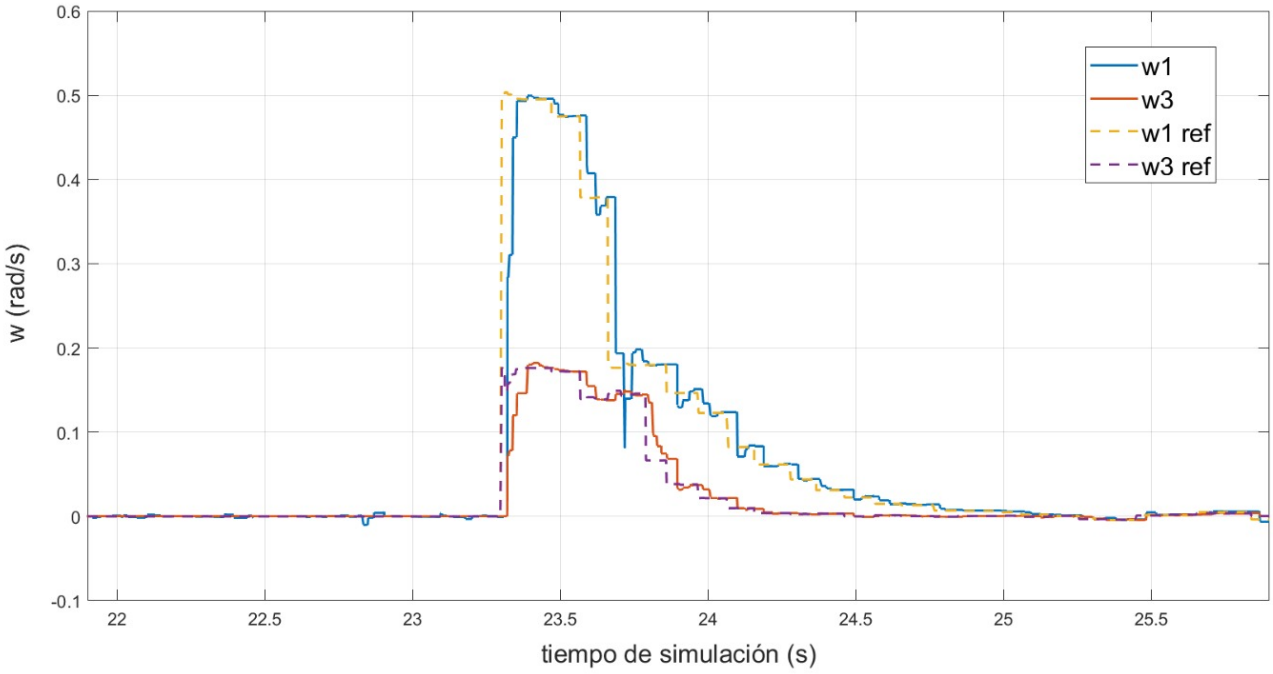

Figura 6.3 Seguimiento de las velocidades angulares de referencia para el gimbal (prueba 2).

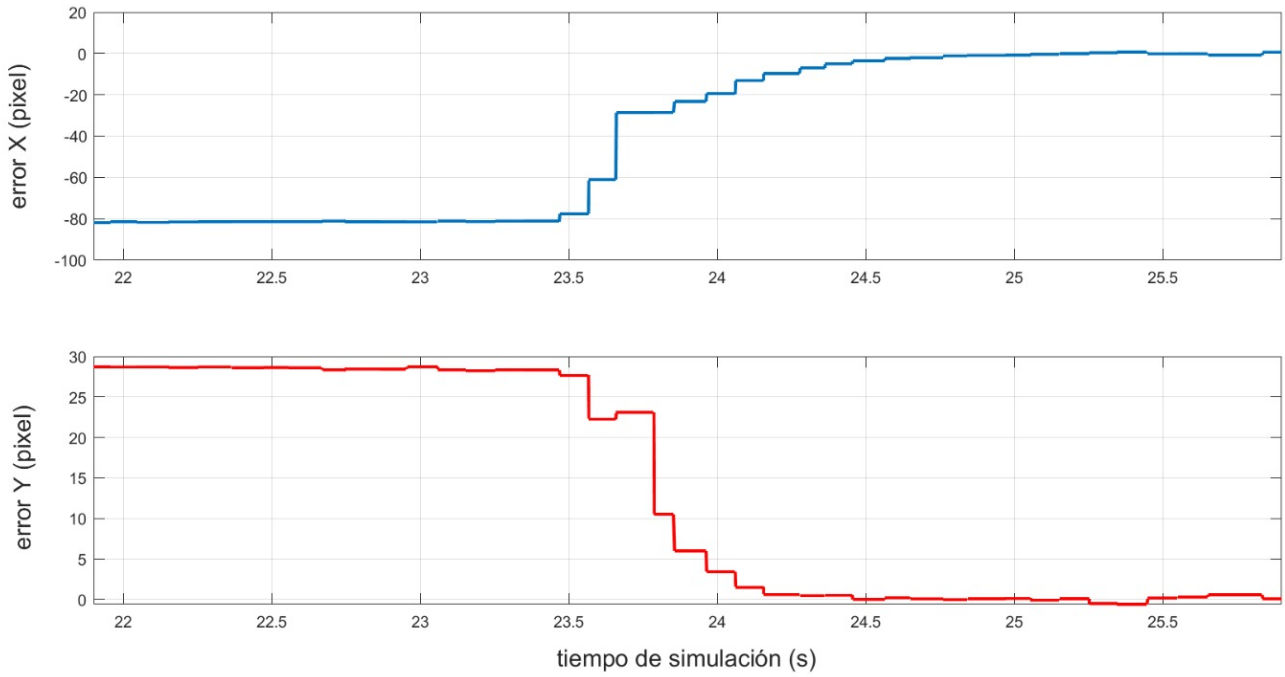

Figura 6.4 Error respecto al centro de la cámara (prueba 2).

Se puede observar cómo en el caso de los objetivos estáticos se consigue centrarlos en la cámara con éxito. El seguimiento de las referencias cumple con los tiempos de subida especificados.

Uno de los problemas con los que nos podemos encontrar en estos casos es que el objetivo elegido tenga pocos puntos característicos o que alguno de sus puntos característicos coincida con algún otro punto de la escena, lo que a veces hace que varíen las coordenadas de las esquinas que encuadran al objetivo y el centro del objeto cambia bruscamente por un instante.

También aparecen problemas de detección y errores cuando el objetivo está muy alejado de la cámara, lo que dificulta el *matching* de los *keypoints*.

Por tanto, se consiguen los mejores resultados cuando:

- El aspecto del objetivo tiene muchas peculiaridades, lo cual aumenta el número de *keypoints*.
- La cámara está cerca del objetivo en cuestión, facilitando la correspondencia de los puntos clave.

6.2. Seguimiento de un objetivo en movimiento

Velocidad del objetivo: 2 m/s

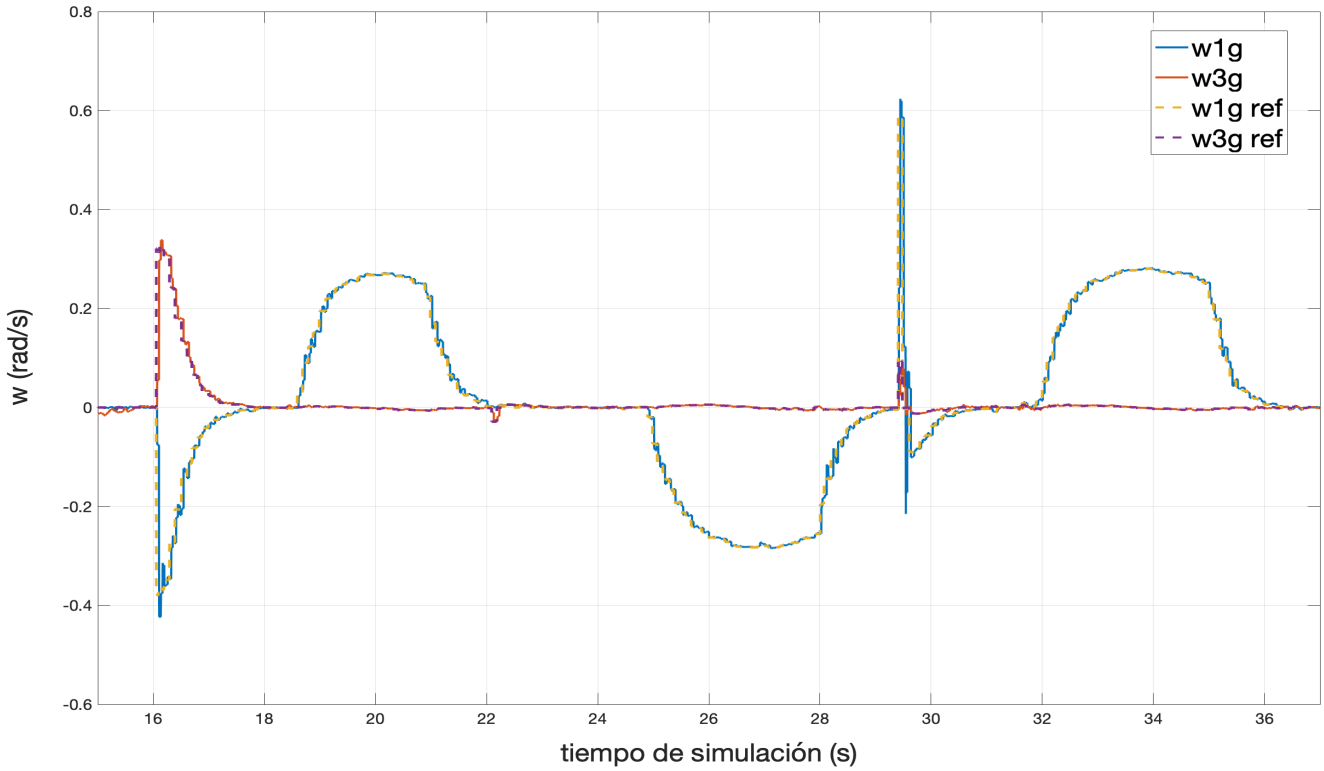

Figura 6.5 Seguimiento de las velocidades angulares de referencia para el gimbal (prueba 3).

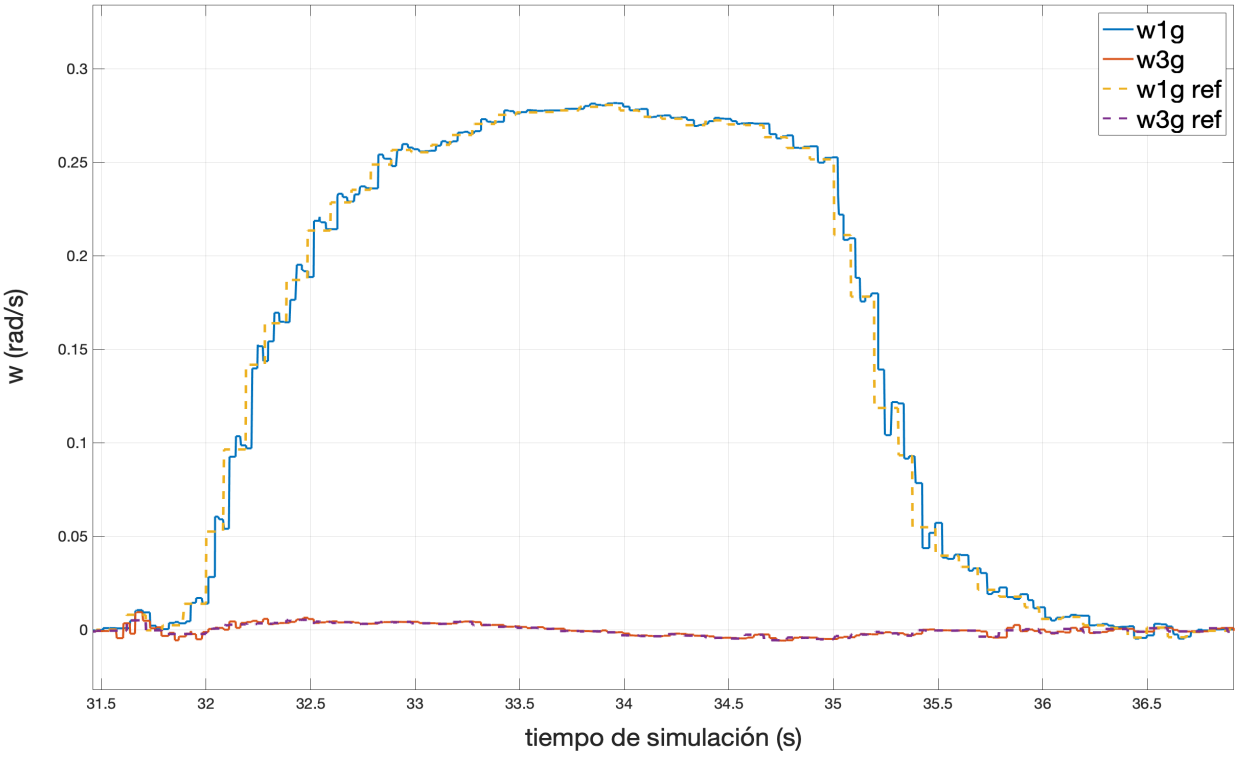

Figura 6.6 Detalle de la figura anterior (prueba 3).

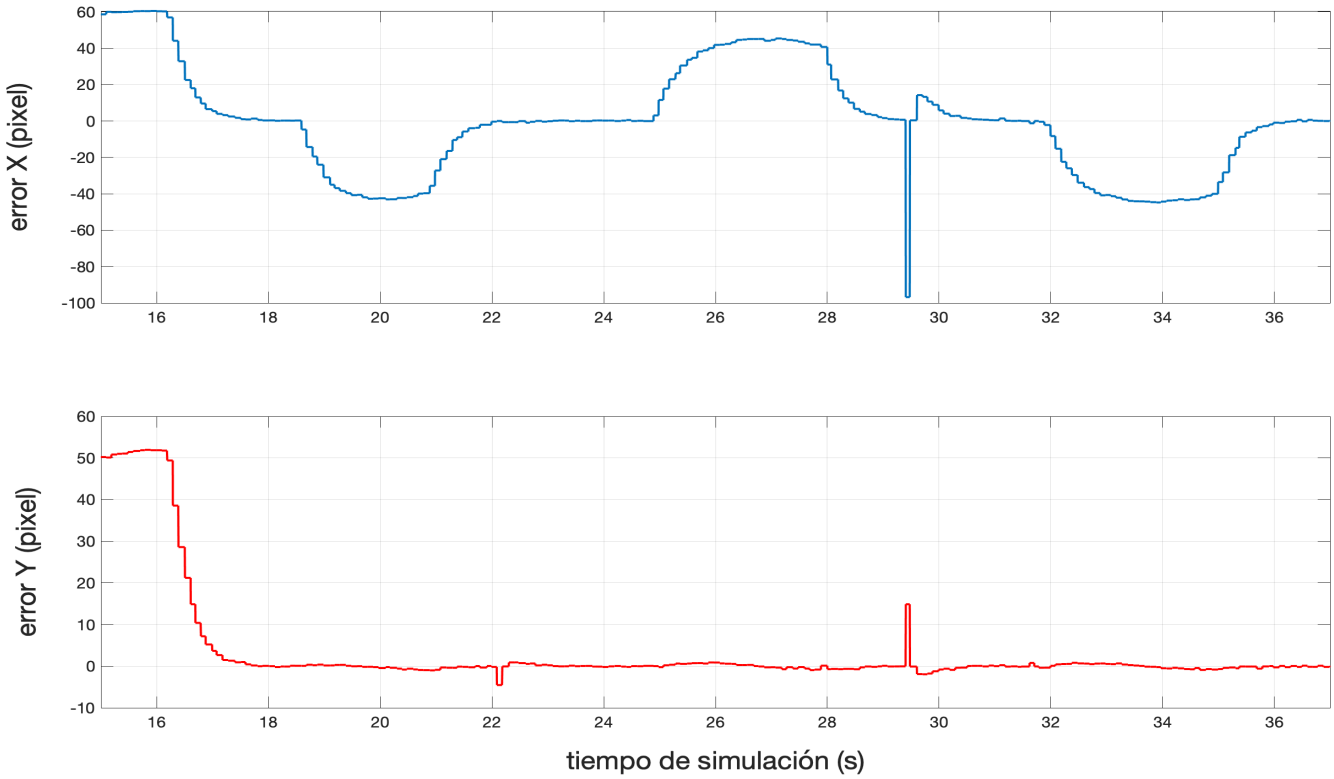

Figura 6.7 Error respecto al centro de la cámara (prueba 3).

En este experimento el objetivo utilizado ha sido el modelo de la camioneta, que se encuentra en la parte inferior derecha de la imagen en un principio.

1. Al principio de la prueba se centra el objetivo en la cámara.
2. El objetivo empieza a moverse hacia la izquierda y se detiene unos segundos.
3. El objetivo se desplaza hacia el otro extremo de la carretera y se vuelve a detener.
4. Por último, regresa al extremo izquierdo y se detiene.

Como se comentó en el capítulo 2, no se predice la trayectoria del objetivo, por lo que cuando este está en movimiento nunca va a estar en el centro de la cámara, pero si se va a seguir su trayectoria.

Los intervalos en los que los errores de las coordenadas son nulos corresponden a los instantes en los que la camioneta se encuentra parada en uno de los extremos.

También se observa cómo sobre el segundo 29 de la simulación se produce un pico en el error. Se observa perfectamente en la Figura 6.8. En ese breve instante se detectan erróneamente las esquinas que rodean al objetivo, por lo que el centro de la cámara varía. Esto hace que aumente el error, lo que provoca a su vez un pico en las velocidades angulares de referencia.

Figura 6.8 Error en la estimación del centro del objetivo.

Velocidad del objetivo: 15 m/s

Si aumentamos la velocidad, la cámara continúa siguiendo al objetivo, aunque cuanto mayor sea dicha velocidad, mayor es la dificultad para conseguirlo.

Se ha realizado una prueba con la camioneta a una velocidad de 15 m/s, donde se puede observar que la alta velocidad provoca que el algoritmo de detección tenga fallos instantáneos que perjudican al control.

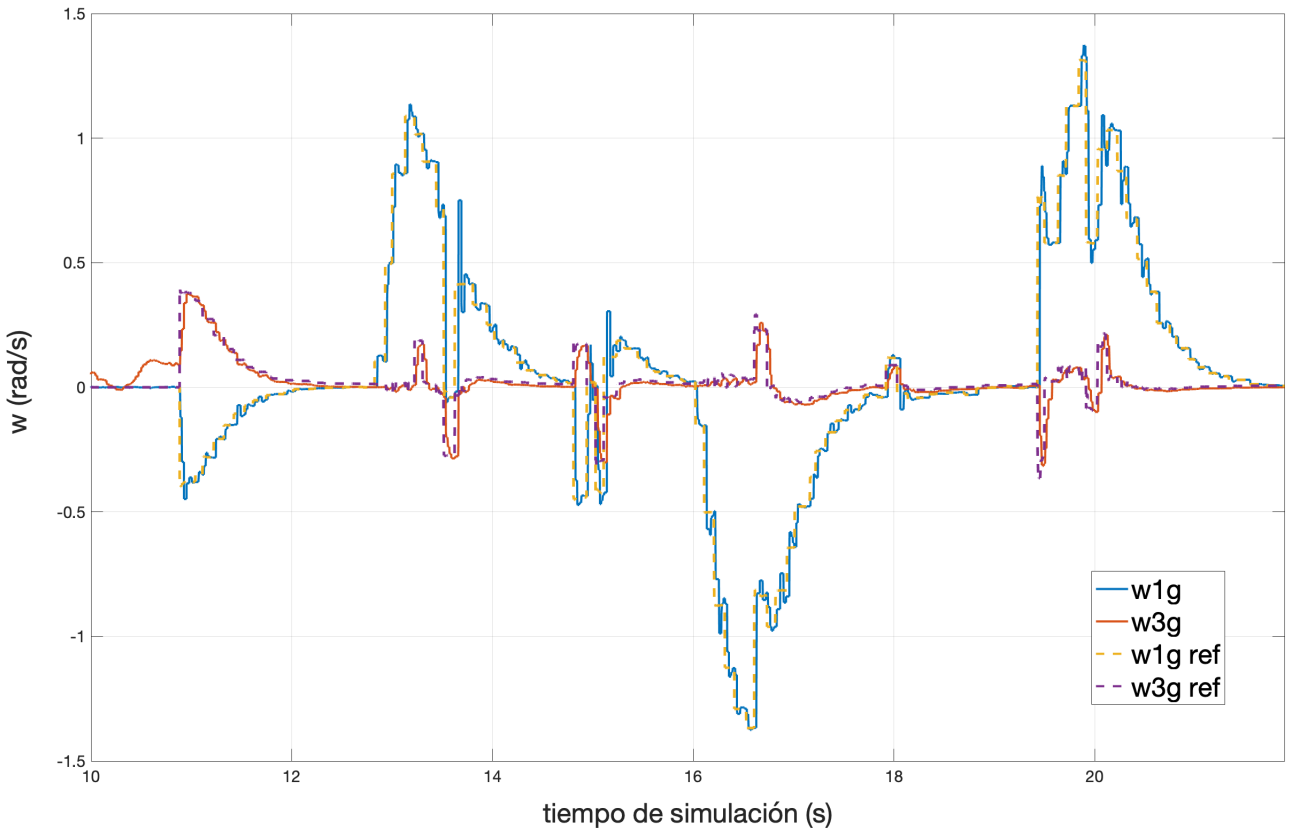

Figura 6.9 Seguimiento de las velocidades angulares de referencia para el gimbal (prueba 4).

En esta figura se pueden observar los numerosos fallos en la detección. El objetivo en un momento dado llega a desaparecer por completo de la imagen, pero debido a que se detiene en un cierto punto la cámara es capaz de seguirlo.

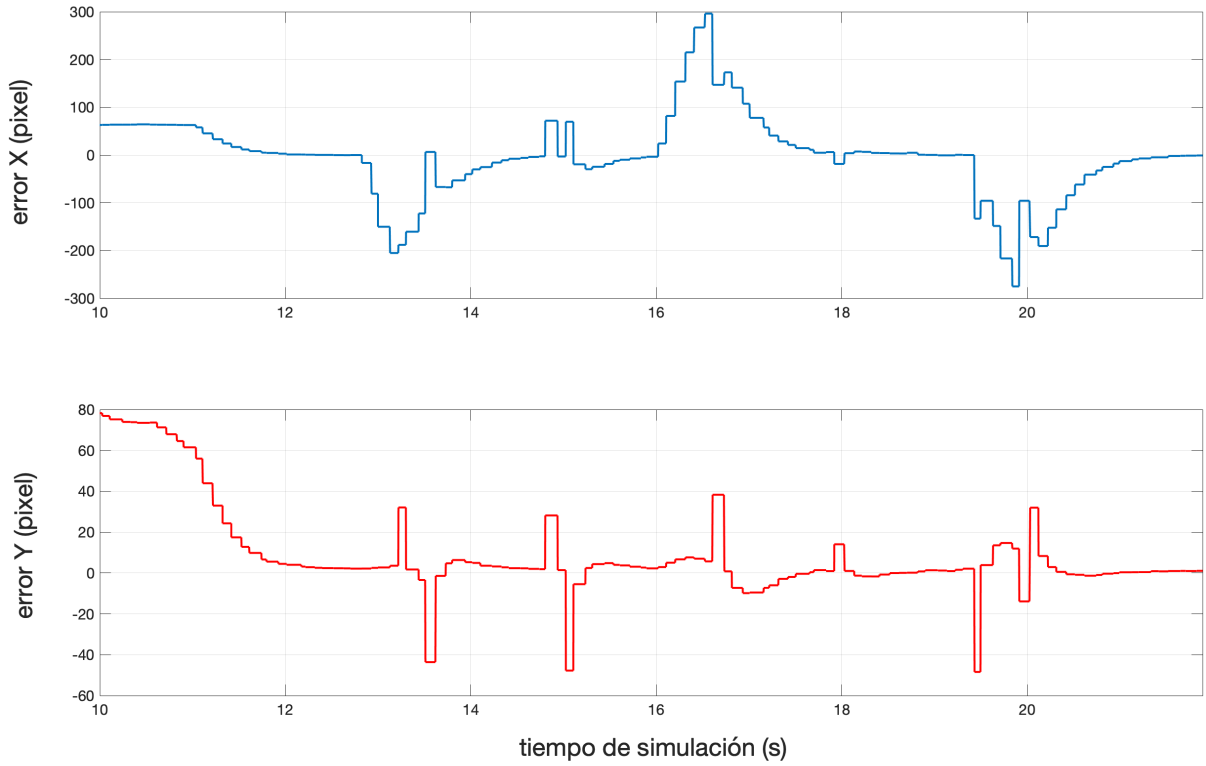

Figura 6.10 Error respecto al centro de la cámara (prueba 4).

Todos los picos que aparecen en esta figura se corresponden con los errores en la detección de la camioneta, provocando pequeños giros bruscos de la cámara.

Velocidad del objetivo: 30 m/s

Con esta prueba se buscaba ver la velocidad del objetivo que provocaba el fallo total del seguimiento para este tipo de experimento.

En las figuras mostradas en la página siguiente se observan los abundantes picos resultantes del error del algoritmo de detección, llegando en el instante final al fallo, perdiéndose el objetivo fuera de la imagen de la cámara.

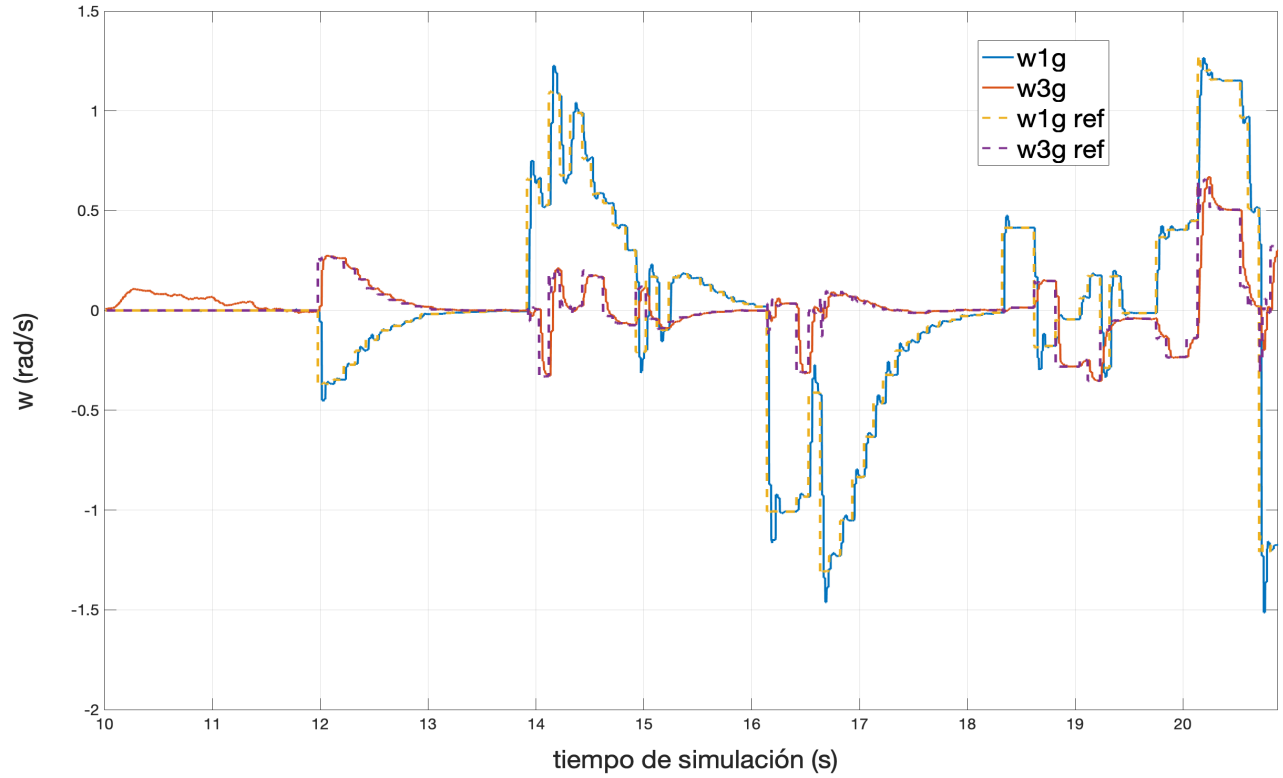

Figura 6.11 Seguimiento de las velocidades angulares de referencia para el gimbal (prueba 5).

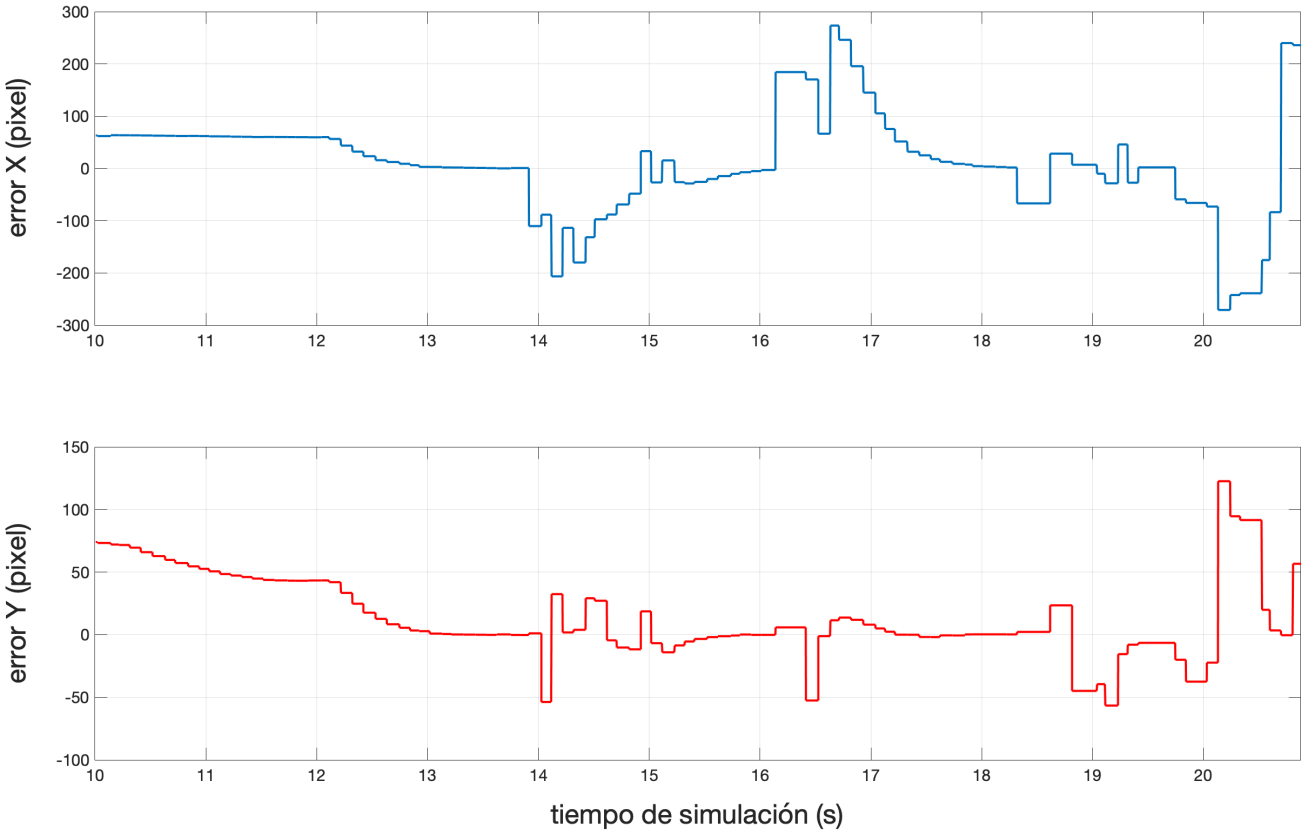

Figura 6.12 Error respecto al centro de la cámara (prueba 5).

6.3. Seguimiento de un objetivo estático con dron en movimiento

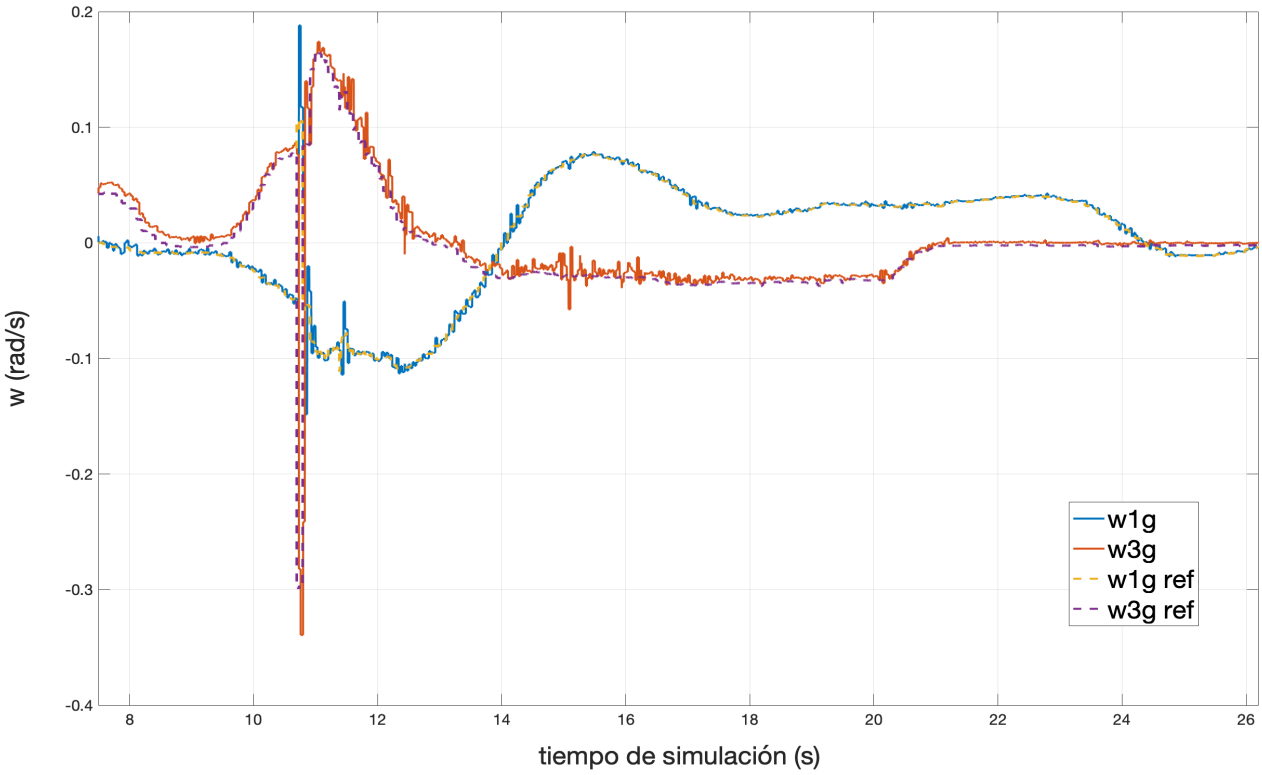

Figura 6.13 Seguimiento de las velocidades angulares de referencia para el gimbal (prueba 6).

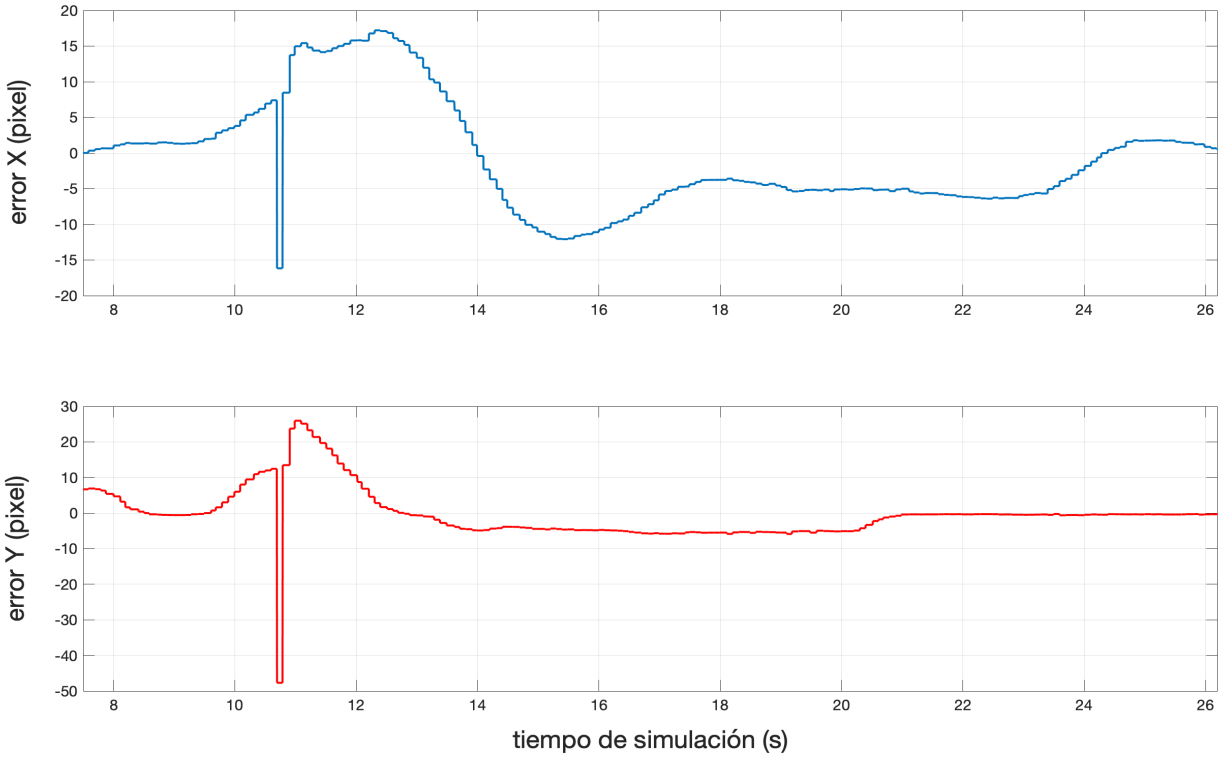

Figura 6.14 Error respecto al centro de la cámara (prueba 6).

En esta prueba el dron se encuentra en movimiento siguiendo una cierta trayectoria cambiante.

Como el movimiento del dron es lento debido a sus limitaciones, el seguimiento se cumple sin ningún problema, aunque el objetivo no aparece del todo centrado en la cámara debido al los movimientos realizados por el cuerpo del dron.

7 CONCLUSIÓN Y FUTUROS TRABAJOS

Este proyecto, además del desarrollo realizado, tuvo inicialmente una fase larga de aprendizaje acerca del trabajo original, en la que se logró entender toda su amplia estructura y familiarizarse con la simulación y el funcionamiento de los nodos del proyecto.

Como conclusión de este trabajo y tras observar los experimentos realizados, podemos afirmar que finalmente se ha conseguido introducir un control para realizar el seguimiento de objetivos al sistema inicial del dron del que partíamos. Pero siempre podemos encontrar mejoras en el trabajo.

El problema del control en velocidad del gimbal funciona perfectamente, siguiéndose las referencias sin problema y ante cualquiera de las pruebas propuestas. El posible problema que hace que el seguimiento del objetivo falle reside en el algoritmo de detección, cuyo funcionamiento podría mejorarse con la integración de algún tipo de filtrado para los puntos mal detectados en las imágenes.

Futuros trabajos

Sobre este proyecto pueden seguir desarrollándose múltiples aplicaciones y mejoras, tales como:

- Un filtrado para el algoritmo de detección, para eliminar los errores instantáneos que provocan giros bruscos en la cámara.
- Se puede buscar también un método de seguimiento basado en inteligencia artificial, adquiriendo información acerca del comportamiento del objetivo para posteriormente predecir su trayectoria y poder identificarlo correctamente para obtener óptimos resultados.
- Seguimiento del objetivo posicionalmente con el dron: conseguir que el dron se mantenga siempre a la misma distancia del objetivo, es decir, perseguirlo en todo momento para nunca perderlo en la imagen de la cámara.

Este trabajo ha significado un largo trayecto de aprendizaje para mí. Me ha enseñado a que no hay que desesperar, aunque las cosas no funcionen en algún momento. Todo con paciencia sale mejor.

APÉNDICE A: CÓDIGOS DE ROS

A.1 Cálculo de la ley de control del gimbal

Código A.1 Script *calculo_wref.py*.

```
#!/usr/bin/env python
######################################################################
# Autor:  Iván de la Fuente Trinidad
# Fecha:  Julio 2021
#
# calculo_wref.py
# Este script calcula la ley de control descrita en el paper
# "Gimbal Control for Vision-based Target Tracking" que obtiene las velocidades
#  angulares de referencia del gimbal para el seguimiento del objetivo
######################################################################

# Librerías
import rospy
from image_pose_estimation.msg import target_cam_pose_msg
from ctrl_desacoplado.msg import calc_wref_msg
from sensor_msgs.msg import Imu
import cv2
import numpy as np

# Inicialización de variables
centre_x = 0.0
centre_y = 0.0
a = np.array([0,0,0])
A = np.array([[315.81579288594037, 0.0, 265.5],
              [0.0, 315.81579288594037, 150.5],
              [0.0,               0.0,   1.0]])
A_1 = np.linalg.inv(A)
k = 1

# Función que calcula la omega de referencia (wref) y la publica
def wref_publishing():
    wref_msg = calc_wref_msg()
    y = np.array([centre_x, centre_y, 1])
```

```
    q = np.dot(A_1,y)/np.linalg.norm(np.dot(A_1,y))
    Sq = np.array([[  0,   -q[2],   q[1]],
                   [ q[2],     0,   -q[0]],
                   [-q[1],   q[0],     0]])

    col1 = -np.dot(np.dot(Sq,Sq),a)/np.linalg.norm(np.dot(np.dot(Sq,Sq),a))
    col2 = np.dot(Sq,a)/np.linalg.norm(np.dot(Sq,a))
    col3 = q
    Re = np.array([[col1[0],  col2[0], col3[0]],
                   [col1[1], col2[1], col3[1]],
                   [col1[2], col2[2], col3[2]]])
    Re_ReT = Re-np.transpose(Re)

    if np.isnan(np.sum(Re_ReT)) == False and centre_x != 0:  # Para asegurarnos que no publica hasta que el nodo que
                                                             # identifica al objetivo haya detectado al objetivo
        S_1 = [Re_ReT[2][1], Re_ReT[0][2], Re_ReT[1][0]]
        wref = [-k*S_1[0], -k*S_1[1], -k*S_1[2]]    # Ley de control del gimbal
        # print(wref)
        wref_msg.wref = wref
        wref_pub.publish(wref_msg)

# Posición del centro del target
def callback_target_cam_pose(data):
    global centre_x, centre_y
    centre_x = data.centre_x
    centre_y = data.centre_y

# Datos del acelerómetro de la IMU del Gimbal2
def callback_accel_imu(data):
    global a
    a = np.array([data.linear_acceleration.z, data.linear_acceleration.y, -data.linear_acceleration.x])

if _name_ == '_main_':
    try:
        rospy.init_node('calculo_wref', anonymous=True)
        # Conexion al Topic mediante suscripción: lectura del centro del objetivo en la cámara
        rospy.Subscriber("/target_cam_pose", target_cam_pose_msg, callback_target_cam_pose)
        # Conexion al Topic mediante suscripción: lectura de aceleraciones de la gravedad del plugin imu
        rospy.Subscriber("/dron/imu_gimbal", Imu, callback_accel_imu)
        # Publicador
        wref_pub = rospy.Publisher("/parametros/wref",calc_wref_msg,queue_size=1)
```

```
        while not rospy.is_shutdown():
            rate = rospy.Rate(500)
            wref_publishing()
            rate.sleep()

    except rospy.ROSInterruptException:
        pass
```

A.2 Control del gimbal en velocidad

Código A.2 Control del gimbal en velocidad. Modificación del modo de funcionamiento normal de *Control_Gimbal.cpp*.

```
/* Modo de funcionamiento normal: controladores PID */
    q1f_refk = q1_ref;
    q2f_refk = q2_ref;
    q3f_refk = q3_ref;

    //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    /* PID de q1 */
    if(tracking1)
    {
     if(wref[1] == 0.0)
       ekq1 = q1f_refk - q1;
     else
     {
       ekq1 = k1*(wref[1]-w1);
       aq1 = P2[0] * (1 + Ts*I2[0] + D2[0]/Ts);
       bq1 = P2[0] * (- 1 - 2*D2[0]/Ts);
       cq1 = P2[0] * D2[0]/Ts;
     }
    }
    else  ekq1 = q1f_refk - q1;
    //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    ukq1 = ukq1_1 + aq1*ekq1 + bq1*ekq1_1 + cq1*ekq1_2;
    ukq1_1 = ukq1;
    ekq1_2 = ekq1_1;
    ekq1_1 = ekq1;
```

```
    /* PID de q2: se mantiene el control en posición */
    ekq2 = q2f_refk - q2;
    ukq2 = ukq2_1 + aq2*ekq2 + bq2*ekq2_1 + cq2*ekq2_2;
    ukq2_1 = ukq2;
    ekq2_2 = ekq2_1;
    ekq2_1 = ekq2;

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    if(tracking3)
    {
     /* PID de q3 */
     if(wref[0] == 0.0)
       ekq3 = q3f_refk - q3;
     else
     {
       ekq3 = k3*(w3-wref[0]);
       aq3 = P2[2] * (1 + Ts*I2[2] + D2[2]/Ts);
       bq3 = P2[2] * (- 1 - 2*D2[2]/Ts);
       cq3 = P2[2] * D2[2]/Ts;
     }
    }
    else   ekq3 = q3f_refk - q3;
    //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    ukq3 = ukq3_1 + aq3*ekq3 + bq3*ekq3_1 + cq3*ekq3_2;
    ukq3_1 = ukq3;
    ekq3_2 = ekq3_1;
    ekq3_1 = ekq3;
```

A.3 Movimiento del modelo *pickup*

Código A.3 Script *mueve_pickup.py*.

```
#!/usr/bin/env python
from gazebo_msgs.srv import GetModelState
from gazebo_msgs.msg import ModelState
import time
import rospy
x = 0
y = 0
z = 0
```

```
model_name = 'pickup'

def show_gazebo_models():
    global x,y,z
    vel = 2.0           # m/s
    freq = 200.0        # Hz
    lim = 4.0
    aux = 0
    try:
        pub = rospy.Publisher('/dron/gazebo/set_model_state', ModelState, queue_size=1)
        rospy.init_node('pickup_movement', anonymous=True)
        rate = rospy.Rate(freq)
        while not rospy.is_shutdown():
            model_coordinates = rospy.ServiceProxy('/dron/gazebo/get_model_state', GetModelState)
            resp_coordinates = model_coordinates(model_name,'link')
            move_msg = ModelState()
            x = resp_coordinates.pose.position.x
            y = resp_coordinates.pose.position.y
            z = resp_coordinates.pose.position.z
            print("Coordinates of "+model_name+ " model:")
            print("x : " + str(x))
            print("y : " + str(y))
            print("z : " + str(z))
            print("---")

            move_msg.model_name = model_name
           ############################################################
            # Trayectoria lineal:
            move_msg.pose.position.x = x
            if y > 4:
                aux = 1
                time.sleep(6)
            if y < -4:
                aux = 0
                time.sleep(6)
            if aux:
                move_msg.pose.position.y = y - (1/freq)*vel
            else:
                move_msg.pose.position.y = y + (1/freq)*vel
```

```
            move_msg.pose.position.z = 0
            move_msg.pose.orientation.x = resp_coordinates.pose.orientation.x
            move_msg.pose.orientation.y = resp_coordinates.pose.orientation.y
            move_msg.pose.orientation.z = resp_coordinates.pose.orientation.z
            move_msg.pose.orientation.w = resp_coordinates.pose.orientation.w
            ############################################################

            pub.publish(move_msg)
            rate.sleep()



    except rospy.ServiceException as e:
        rospy.loginfo("Get Model State service call failed:  {0}".format(e))



if __name__ == '__main__':
    try:
        show_gazebo_models()
    except rospy.ROSInterruptException:
        pass
```

REFERENCIAS

[1] JÁÑEZ VAZ, Isidro Marcelo, 2020. *Modelado y Simulación de un Quadrotor con Gimbal mediante Gazebo y ROS*. Trabajo Fin de Grado Inédito. Universidad de Sevilla, Sevilla.

[2] CUNHA, Rita, et al. *Gimbal control for vision-based target tracking*. En Proceedings of the European Conference on Signal Processing. 2019.

[3] BORREGO PRADO, Fernando Manuel, 2020. *Modelado y Control de un Quadrotor perior de Ingeniería con Gimbal mediante la perspectiva Fly-The-Camera*. Trabajo Fin de Grado Inédito. Universidad de Sevilla, Sevilla.

[4] Colaboradores de Wikipedia. Vehículo aéreo no tripulado. *Wikipedia, La enciclopedia libre* [en línea]. 2021 [fecha de consulta: 13 de septiembre del 2021]. Disponible en <https://es.wikipedia.org/w/index.php?title=Veh%C3%ADculo_a%C3%A9reo_no_tripulado&oldid=137413486>.

[5] ALEGRE GUTIÉRREZ, Enrique, et al. *SIFT (Scale Invariant Feature Transform)*.

[6] Tutorial: Beginner: Intro. *Gazebo* [en línea]. [sin fecha] [fecha de consulta: 7 de agosto de 2021]. Disponible en: http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b1.

[7] About ROS. *ROS.org* [en línea]. [sin fecha] [fecha de consulta: 7 de agosto de 2021]. [sin fecha] Disponible en: https://www.ros.org/about-ros/.

[8] HESS, Rob. SIFT Features. *OpenSIFT: An Open-Source SIFT Library* [en línea]. [sin fecha] [fecha de consulta: 10 de agosto de 2021]. Disponible en: https://robwhess.github.io/opensift/.

[9] Introduction to SIFT (Scale-Invariant Feature Transform). *OpenCV* [en línea]. [sin fecha] [fecha de consulta: 10 de agosto de 2021]. Disponible en: https://docs.opencv.org/4.5.1/da/df5/tutorial_py_sift_intro.html.

[10] YILMAZ, Alper; JAVED, Omar; SHAH, Mubarak. *Object tracking: A survey*. Acm computing surveys (CSUR), 2006, vol. 38, no 4, p. 13-es.

[11] Nodos, Topics y Mensajes. *ROS TUTORIAL* [en línea]. [sin fecha] [fecha de consulta: 13 de agosto de 2021]. Disponible en: http://rostutorial.com/4-nodos-topics-y-mensajes-turtlesim/.

[12] PETROVICHEVA, Anna. Multiple Object Tracking in Realtime. *OpenCV* [en línea]. 27 de octubre de 2020 [fecha de consulta: 16 de agosto de 2021]. Disponible en: https://opencv.org/multiple-object-tracking-in-realtime/.

[13] CHALLA, Subhash, et al. *Fundamentals of object tracking*. Cambridge University Press, 2011.

[14] TYAGI, Deepanshu. Introduction to SIFT. *Medium* [en línea]. 16 de marzo de 2019 [fecha de consulta: 14 de agosto de 2021]. Disponible en: https://medium.com/data-breach/introduction-to-sift-scale-invariant-feature-transform-65d7f3a72d40.

[15] How to Move a Gazebo Model from Terminal. *VarHowto* [en línea]. 15 de mayo de 2020 [fecha de consulta: 23 de agosto de 2021]. Disponible en: https://varhowto.com/how-to-move-a-gazebo-model-from-command-line-ros/.

[19] HERRERO, Fernando. iri_imu_gazebo. *GitLab* [en línea]. 11 de febrero de 2021 [fecha de consulta: 15 de agosto de 2021]. Disponible en: https://gitlab.iri.upc.edu/labrobotica/ros/sensors/imu/iri_imu_gazebo/-/tree/master.

[20] Tutorial: Make an animated model (actor). *Gazebo* [en línea]. [sin fecha] [fecha de consulta: 20 de agosto de 2021]. Disponible en: http://gazebosim.org/tutorials?tut=actor&cat=build_robot.

[21] DMITRY, Devitt. image_pose_estimation. *GitHub* [en línea]. 21 de marzo de 2019 [fecha de consulta: 9 de junio de 2021]. Disponible en: https://github.com/GigaFlopsis/image_pose_estimation.

[22] LENTIN, Aleena. What is ROS? *Robocademy* [en línea]. 1 de julio de 2020 [fecha de consulta: 11 de agosto de 2021]. Disponible en: https://robocademy.com/2020/07/01/what-is-ros/.

GLOSARIO

UAV: Unmanned Aerial Vehicle 4

VANT: Vehículo Aéreo No Tripulado 4

RPAS: Remotely Piloted Aircraft System 4

ROS: Robot Operating System 4

IMU: Inertial Measurement Unit 4

SIFT: Scale Invariant Feature Transform 4

PID: Proporcional Integral Derivativo 4

PI: Proporcional Integral 4

GPS: Global Positioning System 4